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

CUSTOMIZABLE GRADUAL EFFECTS FOR SCALA

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

MATÍAS TORO IPINZA

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
PABLO BARCELÓ BAEZA
ALEXANDRE BERGEL
ISMAEL FIGUEROA PALET

Este trabajo ha sido financiado por CONICYT-PCHA/Magíster Nacional/2013-22131048

SANTIAGO DE CHILE
2015

Resumen

Operaciones realizadas por un programa de computación pueden producir efectos. Efectos computacionales pueden ser definidos como operaciones que interactúan y que se comunican con su ambiente. Ejemplos de efectos son imprimir en pantalla, leer data de usuarios, asignación de memoria, excepciones, etc. Una función que no produce efectos es llamada una función *pura*. Los sistemas de efectos ayudan a controlar estos "efectos colaterales". Por ejemplo, permite correr funciones puras en paralelo sin temer a obtener carreras de datos. Una function pura también preserva la transparencia referencial, asegurando que si una función es aplicada con los mismos argumentos mas de una vez, el resultado sigue siendo el mismo.

En este trabajo se diseñó y desarrolló un sistema de efectos genérico que se caracteriza por ser práctico, combinando para ello tres conceptos de efectos fundamentales: efectos graduales, efectos polimórficos y efectos personalizados.

Debido a que la verificación estática puede ser demasiada restrictiva (algunos programas válidos son rechazados), Bañados *et al* propusieron una teoría que mezcla chequeo de efectos dinámico y estático, permitiendo a desarrolladores a adaptar "gradualmente" una disciplina de efectos en proyectos existentes. Este sistema de efectos graduales da la flexibilidad para moverse entre un sistema completamente anotado con efectos (estático) a uno sin anotaciones de efectos (dinámico), introduciendo chequeos en tiempo de ejecución cuando sea necesario.

Rytz *et al* diseño e implementó un sistema de efectos polimórficos, el cual incrementa la expresividad de los sistemas de efectos usando patrones de orden superior para efectos. Una función es polimórfica en los efectos de su argumento si es que los efectos de la función depende de los efectos de su argumento. Esto permite expresar funciones como `map`, pura en efectos salvo por los efectos que la función que recibe como argumento pueda producir.

En general los dominios de efectos carecen de localidad: tienen un alcance muy general. Por ejemplo: no es necesario considerar toda operación de entrada/salida como un efecto. Los sistemas de efectos no permiten hacer un seguimiento de efectos de operaciones específicas definidas por el usuario. Incorporar y personalizar un sistema de efectos requiere conocimientos previos acerca de sistemas de efectos. Inclusive, con la experiencia adecuada, implementar una disciplina de efectos puede ser difícil o lento. Es necesario un sistema que permita la fácil creación de dominios de efectos.

El trabajo de ésta tesis consiste en tres partes. La primera parte extiende la formalización del sistema de efectos graduales añadiendo efectos polimórficos. En la segunda parte se diseña la sintaxis y semántica necesaria para crear dominios de efectos a través de un lenguaje específico de dominio (DSL). El DSL permite la creación de dominios de efectos con estructuras y relaciones complejas, y la especificación de donde los efectos deben ser producidos mediante especificaciones de efectos externa. También se entrega una formalización del DSL extendiendo el sistema de efectos combinado de la primera parte. La tercera y final parte consiste en la implementación del sistema final de efectos y del DSL en el lenguaje de programación Scala a través de *plugins* para el compilador.

Abstract

Operations done by computer programs may produce effects. Computational effects may be defined as operations that communicate with their environment and perform changes to it. Examples of effects are printing to screen, reading data from users, allocating memory, throwing exceptions, etc. A function that does not produce effects is called a *pure* function. Effect systems helps to control these "side effects". For example, in parallel programming, pure functions can run in parallel without races. A pure function also preserves referential transparency, assuring that if a function is executed two or more times with the same arguments, it outputs the same result.

In this work, we designed and developed a generic effect system that focuses on being practical, combining three fundamental concepts: gradual effects, polymorphic effects and customizable effects.

Given that static checking is necessarily conservative (some valid programs are rejected), Bañados *et al* proposed a theory that combines static and dynamic effect checking. This help programmers to "gradually" adapt an effect discipline into existing projects. Also, this effect system gives the flexibility to move from a fully effect annotated system (static) into a system without effect annotations (dynamic) and vice-versa, introducing runtime checks only where it is needed.

Rytz *et al* designed and implemented a polymorphic effect system which increases the expressiveness of effect systems by supporting high-order patterns for effects. A function is polymorphic on its arguments if the effects of the function depend on the effects of its arguments. This allows one to declare functions like `map` as pure except for the effects that the function received as arguments may produce.

In most cases, effect domains lack locality: their scope is global. For instance, we do not need to consider every input/output operation as an effectful operation. Effect systems do not allow programmers to indicate which specific effectful operations must be tracked. To create and customize an effect system, programmers must have expertise in effect system and its implementation is not trivial, therefore defining an effect discipline can be hard or slow. It is desirable a system that help programmers to easily create or modify effect disciplines.

The work of this thesis consists in three parts. The first part extends the formalization of gradual effect systems with polymorphic effects. In the second part we design the syntax and semantics to easily tailor effect disciplines through a domain-specific language (DSL). The DSL allows programmers to create effect disciplines that have complex lattices and specifiy where effects must be produced through *external effect specifications*. Also, we formalize the DSL by extending the combined effect system of part one. The third and final part consists in the implementation of the effect system and the DSL as a compiler plugin for the Scala programming language.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computational effects can be defined as operations that communicate with their environment and perform changes to it [11]. Functions have "side-effects" if they have observable effects in the environment [11]. Some examples of such effects are: printing to screen, throwing exceptions or mutating memory cells. Effect systems control these effects.

The advantages of using an effect system discipline are many. For example, in parallel programming, *pure* or effect-free functions can run in parallel without races. In the context of exceptions, an effect system could statically ensure that the program is ready to deal with every exception before the program is run. Restricting the allocation of memory in some programs that run periodically, like a daemon, can ensure that the program does not consume all the memory (memory leaks). A pure function also preserves referential transparency, assuring that if the function is executed two or more times with the same arguments, it outputs the same result.

The only effect system that has been put into practice and widely used on large scale applications is Java checked exceptions [11]. However because of its verbosity and lack expressiveness, programmers frequently bypass it.

The importance of using an effect system is clear, but what would happen if different domains of effects are mixed, for example, printing with exceptions? Marino and Millstein [8] proposed a generic type-and-effect system, which allows to work with multiple domains at the same time. This system works as a privilege checking system, verifying that an expression has the necessary permissions right before performing an effectful operation. This is different from systems that produce the information of the computed effects after the expression is executed.

This work is best explained through the "Effect Cube" shown in figure 1.1:

The effect cube is composed of three axes. The origin is the Generic Effect System of Marino and Millstein. One axis adds Effect Polymorphism, another one adds Gradual Effects and the third one makes the effect system customizable. First we will combine gradual effects with effect polymorphism, then we extend that system with the capability to be

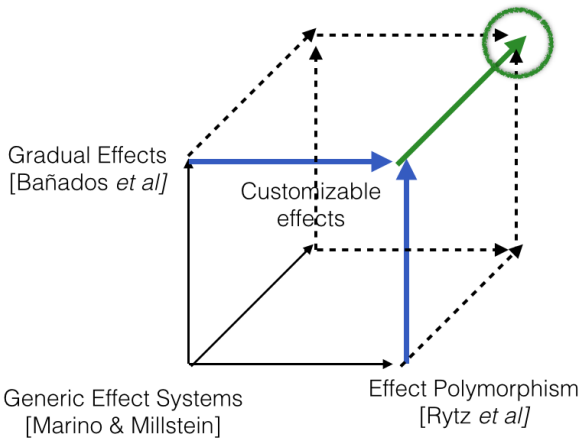


Figure 1.1: The Effect Cube

customizable by joining the three axes together. Finally we implement the solution into the Scala programming language.

The generic system of Marino and Millstein has been adopted by Rytz *et al.* [10], to develop a generic effect system for Scala. The system tries to address the verbosity issue of effect systems by introducing a lightweight syntax to annotate effects. In particular, this system supports effect-polymorphic functions, allowing the concise expression of higher-order programming patterns. The system, contrary to the system of Marino and Millstein, infers the latent effects of a given expression, rather than checking them against a set of held privileges.

The benefits of effect polymorphism in Rytz's effect system are shadowed by the conservativism of static checking. Indeed, the static approximations done by static effect systems imply that some simple programs that should pass are rejected. A solution to this is to do a *gradual* verification.

Recently, Bañados *et al.* [1] proposed a gradual approach to effect checking, i.e., a programmer-controlled migration between run-time type checking and static type checking, where programmer annotations are verified at compile time. This allows programmers to benefit from both kinds of effect verification approaches, and also permits a transition by increments, to a fully statically typed effect discipline or vice versa.

Although generic effect systems exists, existing domains lack locality: their scope is global, and therefore they can be too strict for programmer needs. A programmer may want to track some effectful operations instead of tracking all of them. In other words, effects produced by some operations may be insignificant for the programmer, and he/she may want to quickly tailor or customize an effect discipline to adjust it according to the needs of his/her application. Also, in order to design an effect discipline, an abstract representation for effect must be defined. This abstract representation is different for each effect domain depending on its effect lattice. Sometimes effect lattices present lattices with complex relations thanks to ordering between effects or *sub-effecting*. To adapt an effect domain in the Rytz effect system or the Bañados effect system, the type system must be extended or modified, requiring the

programmer to have a background in type systems and generic effect systems. Therefore, modifying an effect system can sometimes be tedious or cumbersome.

In this thesis, we propose an effect system that focuses on being practical, aiding programmers in easily defining and customizing effect disciplines. This system is impractical if it does not adopt effect polymorphism of the Rytz *et al* effect system and gradual effects of the Bañados *et al* effect system. The practical effect system is implemented as a domain-specific language for the Scala language. Scala is chosen because it is a fast growing language that is widely used in the industry and it is easy to extend.

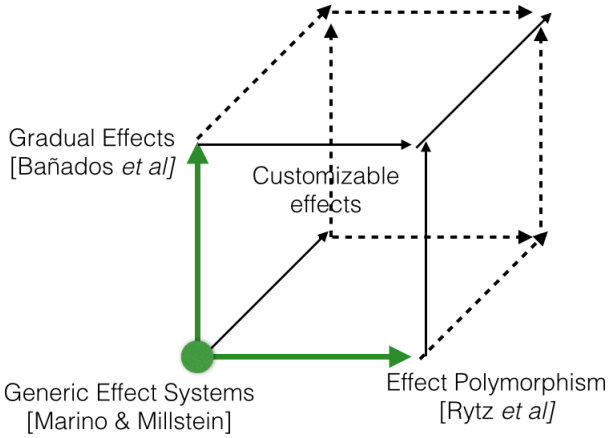
The contributions of this thesis are:

- We design and formalize the syntax and semantics to allow programmers to easily create or modify effect disciplines through a domain-specific language. This help programmers to adapt an effect discipline creating application-specific effect disciplines according to their needs.
- We present a formalization of an effect system that combines gradual effect with effect polymorphism. We extend the gradual effect system proposed in Bañados *et al* with support for effect polymorphism and sub-effecting.
- We implement the practical effect system into the Scala language, being the first implementation of a gradual effect system up to date. The implementation of the type system is fully deterministic by following a bidirectional approach: effect inference and then effect checking. The implementation is developed as Scala compiler plugins and were tested using SBT and the Play framework.

The document is organized as follows: Chapter 2 introduces the background and related work to provide insight into key concepts of this work. It explores effects, effect systems and generic effect systems, gradual type, and briefly explains a gradual effects system presented by Bañados *et al* and a practical effect system for Scala presented by Rytz *et al*. Chapter 3 formalizes a combination of gradual effects with effect polymorphism and sub-effecting (combining two axes of the effect cube). Chapter 4 starts by introducing the motivation on why a domain-specific language is necessary. Next, it explains and presents examples of the domain-specific language, and finalizes the chapter with the formalization of the practical effect system. Chapter 5 presents the implementation of the effect system, explaining why a bidirectional effect system is needed and how it is implemented. It then explains the Scala implementation and its components, and then discusses limitations of the solution. Chapter 6 presents examples and cases of use of the solution.

Chapter 2

Background and Related Work



This chapter gives the background and offers insight necessary to understanding the following chapters. First we familiarize ourselves with effects, effect systems and generic effect systems – our origin on the Effect Cube. Second, we move on one axis of the Effect Cube by introducing gradual typing and then briefly explain the theory about the gradual effects system presented by Bañados *et al*. We finish this chapter by moving on a second axis of the Effect Cube, reviewing the practical effect system for Scala presented by Rytz *et al*. We assume familiarity with this concept throughout the rest of this work.

2.1 Effects

A computer program can have different "side-effects" or interactions with its environment, for example I/O printing:

```
println(''Hello World!")
```

error handling:

```
throw new EmptyStackException();
```

or memory updates:

```
var x: Int = 1
x = 2
```

among others.

The problem with effects is that they break equational reasoning about programs, meaning for example, that evaluating an expression twice may yield different results and effects. In other words, with effects the equivalence between an expression and its value is lost. This produces a situation where certain optimizations can not be done, like inlining of constants or pre evaluation of expressions. It also makes the task of reasoning about programs much more difficult.

Effects are uncontrolled: as a matter of fact, most languages permit the execution of effects everywhere and every time. To regain some reasoning capabilities, it would be useful to control where effects can be performed, for example by defining that a function is pure (no effects can be produced), then it will not be able to print to the screen.

2.2 Effect Systems

Effect systems allow programmers to control the side-effects that expressions may produce. The effects of an expression that are not yet evaluated are called *latent effects*.

Using an effect systems discipline can offer a number of advantages in different contexts. For example, races can be avoided in parallel computing by prohibiting the use of effects in certain parts of the program, or by only permitting local effects. In the exception domain, it can be statically ensured that the program is ready to deal with every exception before the program is run. Also, the referential transparency of an expression can be ensured by restricting its latent effects.

One can design a system that dynamically controls the effects of expressions through a monitor. At runtime, this monitor checks to see if an expression that is about to produce an effect has permissions to proceed. This dynamic approach to effect checking is followed in *computational contracts*, prohibiting or ensuring that certain functions are called, checking access permissions, checking time or memory constraints, etc. all at run time [**?**].

One can enforce an effect discipline without having to execute the program. Sometimes

it is too late to realize that the program is about to do something wrong, and it would have been better to tell and provide support to the programmer at development time. The most common way of checking effects statically is to combine effect checking with type checking. These systems are called type-and-effect systems [6].

In a traditional type system a function can be typed $T_1 \longrightarrow T_2$, where T_1 is the input type and T_2 is the output type. The type system can be extended to include the latent effects that may be produced when the function is applied: $T_1 \xrightarrow{e} T_2$. For instance, let us consider a function that receives an `Int` and returns an `Int`, and that also may print to the screen. This function is typed as $\texttt{Int} \xrightarrow{\{io\}} \texttt{Int}$, where i*o* denotes the effect of printing to screen. Applying such a function in a context that requires pure computation, produces a static effect error.

Similar to static type checking versus dynamic type checking, a static effect system is more complex and restrictive because it is necessarily conservative (some valid programs are rejected). But the final program is more efficient because it does not need to do checks at run time to enforce the effect discipline.

Generic Effect System Marino and Millstein [8] proposed a way to generalize effect systems, i.e., the capability of working with multiple domains of effects at the same time, to avoid "re-inventing effects" for each possible effect discipline.

This is based on a checking system where effects are considered as "privileges": an authorization to produce a particular effect. This effect system works thanks to two functions: **adjust** and **check**. The **adjust** function is in charge of adjusting the set of privileges, i.e., granting and revoking privileges for a given sub-expression and a given a set of "privileges". The **check** function is in charge of checking if an expression has enough privileges to be well typed given a set of "privileges". This system is called a *generic effect system*.

Formally, a type-and-effect judgment has the form $\Phi; \Gamma \vdash \mathrm{e} : T$, where Φ is a set of privileges, indicating what effects are allowed to occur for a given sub-expression e, Γ is the typing environment and T is the type of e. This follows a restriction model, or more technically, a top–down approach of privilege checking, adjusting the privileges for each sub-expression, checking permissions "on the way down". For instance $\{\texttt{@write}, \texttt{@read}\}; \Gamma; \overline{f} \vdash \mathrm{e} : T$ allows the expression e for the effects `@write` and `@read` to be produced. On the other hand, $\emptyset; \Gamma; \overline{f} \vdash \mathrm{e} : T$, means that e can not produce effects.

To better illustrate the system, given the exception domain, if the rule for `try` expression is:

$$\text{T-Try}\frac{\mathbf{adjust}_{TRY}(\Phi)\,; \Gamma \vdash \mathrm{e}_1 : T \qquad \Phi; \Gamma \vdash \mathrm{e}_2 : ExnType \longrightarrow T}{\Phi; \Gamma \vdash \texttt{try}\ \mathrm{e}_1\ \texttt{with}\ \mathrm{e}_2 : T}$$

then the **adjust** function of the e_1 sub-expression expands the given set of privileges to allow the expression to throw exceptions: $\mathbf{adjust}_{TRY}(\Phi) = \Phi \cup \{throws\}$. For the e_2 sub-expression, the privileges set Φ is left unmodified.

Lastly, if the rule for `throw` is:

$$\text{T-Throw}\frac{\Phi;\Gamma \vdash \mathrm{e}\colon ExnType \qquad \mathbf{check}_{TRHOW}(\Phi)}{\Phi;\Gamma \vdash \texttt{throw}\ \mathrm{e}\colon T_1}$$

then when the type checker encounters a `throw` expression within e_1, it must check if *throws* is contained in the set of given privileges to be well typed, so $\mathbf{check}_{THROW}(\Phi) = throws \in \Phi$. Notice that T_1 is not bounded so it can take the place of any type as throwing exception should be possible anywhere.

2.3 Gradual Typing

Static and dynamic type systems have different strengths and weaknesses, and each is well suited for different programming tasks. Many efforts have been made to integrate static and dynamic typing to combine the benefits of both disciplines [5, 2, 9, 7]. However, not much work has been done on type systems that allow programmers to migrate from one discipline to the other. Siek and Taha proposed a solution called *gradual typing* [12] based on the notion of known and unknown types at compile time. It catches inconsistency between these types at run-time. They later on extended this theory for objects [13].

In a gradual type system, statically typed and dynamically typed systems are combined, getting the best of both worlds, i.e., providing the flexibility of dynamically typed languages when program fragments are not annotated with their types, and giving the benefits of statically typed languages when they are [12]. Sections of the program that are annotated are verified at compile-time, and the unannotated sections are verified at run-time. The boundary between statically and dynamically typed parts of the program is addressed by introducing *casts*, which at runtime check that the static assumptions are not violated. If they are violated, a cast error is raised.

Also, the cost of dynamism is "pay-as-you-go", meaning that if a program is fully annotated, it has the same benefits as if it was written in a statically-typed language: early error detection and more efficient program execution. This allows programmers to "gradually" evolve dynamically typed code into statically typed code; and vice versa, a statically typed discipline can be relaxed by selectively introducing dynamicity, giving the full spectrum of possibilities. While a static type system conservatively rejects any program that *may* lead to run-time type errors, a gradual type system is optimistic and accepts all programs that *may not* lead to run-time type errors.

Technically, gradual typing is achieved by extending the type system with a dynamic or unknown type. A new relation between types called *consistency* '$\sim$' is then introduced, defined as:

$$T \sim T \qquad T \sim ? \qquad ? \sim T \qquad \frac{T_1 \sim T_1' \qquad T_2 \sim T_2'}{T_1 \to T_2 \sim T_1' \to T_2'}$$

where '?' represents the unknown type and '$\sim$' represents the consistency relation between types.

In gradual typing, the equality relation (=) in the typing rules is replaced with the consistency relation. Consistency relates static types to the unknown type, in a way that allows the type system to replace the unknown type with any other type and vice versa. It is important to notice that this relation is not transitive, otherwise the property would allow, for example, `Int` to be consistent with `Boolean`, which is undesired.

For example, if functions f and g are defined as:

$f = \lambda x : \texttt{Int. if } y \texttt{ then } x\texttt{+1 else } x$
$g = \lambda x : \texttt{?. if } y \texttt{ then } x\texttt{+1 else } x$

where '`?`' is used to represent the unknown type. Then f(`true`) is rejected by the gradual type system because `Boolean` is not consistent with `Int`. On the contrary, g(`true`) is accepted by the type system because the type of the argument of g is considered unknown and '`?`" is consistent with `Boolean`.

During static checking, casts are inserted at the boundaries between static and dynamic parts of the code, so g is transformed into:

$g = \lambda x : \texttt{?. if } y \texttt{ then <Int>}x\texttt{+1 else } x$
At run-time if y is false, then g(`true`) returns `true`. Otherwise, a run-time cast error is raised (`true` is not an `Int`).

2.4 Theory of Gradual Effects (TGE)

Bañados *et al.* [1] extended the work of Marino and Millstein, resulting in an effect system that is both gradual and generic.

As the system is based on the work of Marino and Millstein, it uses similar functions for checking effects, called **adjust** and **check**. In this work, the set of privileges to be adjusted or checked can contain a dynamic or unknown effect, which is annotated '¿', similar to the symbol for dynamic or unknown types (Section 2.3).

For example, in a system with an IO effect domain, i.e., i*o* privilege is needed before print can be used, if a function f is defined as:

$f = \lambda x : \texttt{Int. if } y \texttt{ then (print(}x\texttt{); 0) else 1}$

then f has the type $\texttt{Int} \xrightarrow{\{io\}} \texttt{Int}$ and can not be used in a context where pure functions are expected. For instance, if $\emptyset$ represents the empty privilege set and h is typed $(\texttt{Int} \xrightarrow{\emptyset} \texttt{Int}) \xrightarrow{\cdots} \texttt{Int}$, then it is impossible to apply f to h, despite y being `false`.

With gradual effects f can be annotated like this:

$f = \lambda x : \texttt{Int}.\ \texttt{if}\ y\ \texttt{then (print}(x)\texttt{; 0)::¿ else 1}$

which is typed $\texttt{Int} \xrightarrow{\{¿\}} \texttt{Int}$, and as ¿ is consistent with $\emptyset$, then the type system accepts the expression $h(f)$. The checking is forced to be at run-time, resulting in an error if the variable y was true.

One of the key insights of that work is to use abstract interpretation [4] to give meaning to ¿. A privilege set that contains ¿ is called a *consistent privilege set*, and represents a number of possible concrete privilege sets. This definition is made precise by the notion of the concretization function γ. For instance, consider a domain of three effects for memory management: `@read`, `@write`, `@alloc`. The concretization of the consistent privilege set $\Xi = \{\texttt{@alloc}, ¿\}$ is the following set of privilege sets:

$\gamma(\Xi) = \{\{\texttt{@alloc}\}, \{\texttt{@alloc}, \texttt{@read}\},$
$\{\texttt{@alloc}, \texttt{@write}\}, \{\texttt{@alloc}, \texttt{@read}, \texttt{@write}\}\}$

The judgment rules, as in Marino and Millstein, are based on adjusting the sets of privileges for the given sub-expressions: $\Xi; \Gamma \vdash \mathrm{e} : T$ where Ξ, instead of being a privilege set, is a consistent privilege set. Then, instead of using standard set containment to relate the produced effects with the permitted ones, the gradual system uses a notion of *consistent containment* between privilege sets, which is roughly set containment modulo the unknown: Ξ_1 is *consistently contained* in Ξ_2, notation $\Xi_1 \sqsubseteq \Xi_2$, if and only if $\Phi_1 \subseteq \Phi_2$ for *some* $\Phi_1 \in \gamma(\Xi_1)$ and $\Phi_2 \in \gamma(\Xi_2)$.

For instance, let us consider the simplified [T-App] rule of the source language:

$$\text{T-App}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma \vdash \mathrm{e}_1 : T_2 \xrightarrow{\Xi_1} T \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi)\,; \Gamma \vdash \mathrm{e}_2 : T_2 \\ \widetilde{\mathbf{check}}_{APP}(\Xi) \qquad T_1 \xrightarrow{\Xi_1} T \lesssim T_2 \xrightarrow{\Xi} T\end{array}}{\Xi; \Gamma \vdash \mathrm{e}_1\ \mathrm{e}_2 : T}$$

To support gradual effects, the [T-App] enhances the Marino and Millstein type for function application by replacing every privilege set with a consistent counterpart. Effect privileges associated with every subexpression are adjusted using the function $\widetilde{\mathbf{adjust}}$ and expressions are checked using the function $\widetilde{\mathbf{check}}$. Effect subtyping is lifted to consistent subtyping which basically applies the notion of *concretization* to the containment operator. For example, $\texttt{Int} \xrightarrow{\{¿\}} \texttt{Int} \lesssim \texttt{Int} \xrightarrow{\{io\}} \texttt{Int}$ holds because $\{¿\}$ can be concretized to a possible set of privileges {io}.

Runtime checks are introduced thanks to two commands: `restrict` and `has`, that adjust and check the dynamic set of effects. The `restrict` command adjusts the set of privileges available in the dynamic extent of the current expression. The `has` expression checks dynamically for privileges.

The source language is translated in order to insert runtime checks. For example [T-App] is translated using the rule:

$$\text{C-App}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon T_2 \xrightarrow{\Xi_1} T \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi)\,;\Gamma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon T_2 \\ \widetilde{\mathbf{check}}_{APP}(\Xi) \qquad \Phi = \Delta_{app}(\Xi) \qquad T_1 \xrightarrow{\Xi_1} T \lesssim T_2 \xrightarrow{\Xi} T\end{array}}{\Xi;\Gamma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \mathit{insert\text{-}has?}(\Phi, (\langle\langle T_2 \xrightarrow{\Xi} T \Leftarrow T_1 \xrightarrow{\Xi_1} T\rangle\rangle \mathrm{e}_1')\mathrm{e}_2') \colon T}$$

In the internal language effectful operations must have enough privileges to be performed, consistent checks are either resolved statically or rely at runtime privileges. When needed, the *insert-has?* inserts a runtime check that verifies the availability of particular effects. The *insert-has?* function only inserts a runtime check when the set of privileges is not empty:

$$\mathit{insert\text{-}has?}(\Phi, \mathrm{e}) = \begin{cases} \mathrm{e} & \texttt{if } \Phi = \emptyset \\ \texttt{has } \Phi\ \mathrm{e} & \texttt{otherwise} \end{cases}$$

The Δ_C function transforms a consistent privilege set into a minimal set of conservative set of additional privileges needed to safely pass the corresponding strict **check** function. Casts are inserted by using the meta-function $\langle\langle T_1 \Leftarrow T_2\rangle\rangle$ only when static subtyping does not already hold.

As is standard in the formalization of gradual typing [12], the semantics of the language is given by translation to an internal language with dynamic checks. The evaluation judgment has the form $\Phi \vdash \mathrm{e} \to \mathrm{e}'$, meaning that e reduces to e′ under the current privilege set Φ. The dynamic operations that are inserted either restrict the current privilege set (`restrict`) or check the current privilege set for a given effect privilege (`has`). These operations are inserted whenever the unknown effect is used in a typing derivation, to enforce the corresponding dynamic checks. If an effect check fails, a runtime effect error is raised. Casts are implemented by introducing `has` and `restrict` commands during evaluation as shown in [E-Cast-Fn]:

$$\Phi \vdash \langle (x_2 : T_{21}) \xrightarrow[x_2]{\Xi_2} T_{22} \Leftarrow (x_1 : T_{11}) \xrightarrow[x_1]{\Xi_1} T_{12}\rangle(\lambda x \colon T_{11}\ .\ \mathrm{e}) \to (\lambda x \colon T_{21}\ .\ \langle T_{22} \Leftarrow T_{12}\rangle \texttt{restrict}\ \Xi_2\ \texttt{has}\ (|\Xi_1|\backslash|\Xi_2|))[(\langle T_{11} \Leftarrow T_{21}\rangle x)/x]\mathrm{e})$$

To illustrate a cast, let us consider a simple example

$$\langle\langle \texttt{Int} \xrightarrow{\{¿\}} \texttt{Int} \Leftarrow \texttt{Int} \xrightarrow{\{\mathrm{io}\}} \texttt{Int}\rangle\rangle(\lambda x \colon \texttt{Int}\ .\ x)$$

a runtime check needeeds to be inserted to check if the privilege `io` is available:

$$(\lambda x \colon \texttt{Int}\ .\ \texttt{has}(\{\texttt{io}\})x)$$

This work allows programmers to gradually introduce effect disciplines in their systems, for example, making their systems dynamic at first, and changing them to a static discipline, only where needed. Bañados *et al.* developed the theory of gradual effects; to the best of our knowledge there is no practical implementation.

2.5 A Practical Effect System for Scala (PES)

Rytz *et al.* recently released an effect type system for Scala [10], a language used in large scale applications. This system extends the work of Marino and Millstein [8] with lightweight effect polymorphism. This work was developed as a compiler plugin and applied to the Scala collections library, which makes extensive use of higher-order functions and effect polymorphism.

Formally, the type-and-effect judgment has the form $\Gamma \vdash \mathrm{e} : T \,!\, \Phi$. The biggest difference between that judgement and the judgement in Marino and Millstein is that it follows an inference model, or more technically, a bottom–up approach of effect inference. This means that the effect system does not work with a set of privileges that is being adjusted and checked. To calculate the effect of the expression e, first the system infers the effects of the sub-expressions of e, and then calculates the effect Φ. The calculated effect Φ must clearly be a subset of the privilege set of an equivalent Marino and Millstein's effect system

Effect-polymorphism A function is considered effect-polymorphic if its *latent* effects depend on the effects of the argument.

This is expressed using *relative effect annotations* on function types, meaning that each function declares a list of parameters which contribute to its effects. Formally, a function type has the form $(f : T) \xrightarrow[\bar{f}]{\mathrm{e}} T$ where e is the *specific* effect and $\bar{f}$ is a list of *relative effects*. The latent effect of a function is the combination of specific and relative effects. For instance, the function type $(f : \texttt{Int} \to \texttt{Int}) \underset{f}{\to} \texttt{Int}$ describes a higher-order function that is polymorphic in the effect of its argument f.

Monomorphic function types are i*mpur*e by default, meaning that they have the largest effect possible for any domain. On the contrary, effect-polymorphic function types are *pur*e by default.

For instance, the higher-order function `map`:

```
def map(l: List[Int], f: Int → Int): List[Int] → (f : Int → Int) ⊥→ List[Int] =
  l match {
    case Nil       => Nil
    case Cons x xs => Cons (f x) (map xs f)
  }
```

applies the function `f` to all elements of the list of Int `l`, meaning that `map` does not introduce new effects, and the only effects that can occur are the effects of its argument `f`.

Annotating methods as effect-polymorphic does not require changing the parameter type of the functions. For example, for the `map` function previously defined, the signature in Scala is `map(l: List[Int], f: Int => Int) @pure(f)`, which indicates that `map` is effect polymorphic on `f`. Relative effects are expanded at call site according to the type of the

argument that is passed for a parameter.

The support for effect-polymorphic functions integrates well in object-oriented languages, where it is very common to find methods that take multiple objects as arguments, and those objects carry a potentially large number of member methods [3]. The example of the `map` function shows that the function has a relative effect `f` and no specific effect, but `f` is an abbreviation of `f.apply`. This means that a function can have relative effects that depend on members of its parameters, for example:

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

def map(l: List[Int], f: Int => Int): List[Int] @pure(f) = {  ... }
```

The function `println` has the relative effects of the method `toString` of its parameter `o` and also produces `@io`. The function `map` has the relative effects of its argument `f` and does not produce additional effects.

To introduce effect polymorphism, formally, the judgment rules take the form $\Gamma; \bar{f} \vdash t : T \,!\, \mathrm{e}$, where $\bar{f}$ is a new environment to indicate on which functions the expression is being polymorphic, and e is the effect inferred for term t.

The inference is done via an **eff** function that combines the different effects of the sub-expressions. To illustrate this function a simplified first version of the typing rule for monomorphic function application is:

$$\text{T-APP-MONO}\ \frac{\Gamma; \bar{f} \vdash t_1 \colon (x : T_1) \xrightarrow[\bar{x}]{\mathrm{e}} T \,!\, \mathrm{e}_1 \qquad \Gamma; \bar{f} \vdash t_2 \colon T_2 \,!\, \mathrm{e}_2 \qquad T_1 <: T_2 \qquad t_1 \notin \bar{f}}{\Gamma; \bar{f} \vdash t_1\ t_2 \colon T \,!\, \mathbf{eff}(APP, \mathrm{e}_1, \mathrm{e}_2, \mathrm{e})}$$

The effects of applying t_1 on t_2 are calculated, thanks to the **eff** function. The **eff** function combines the inferred effects e_1, the specific effect e of the t_1 sub-expression, and the inferred effect of the t_2 sub-expression.

For example, in the `try` expression:

$$\text{T-Try}\ \frac{\begin{array}{cc}\Gamma; \bar{f} \vdash t_1 \colon T_1 \,!\, \mathrm{e}_1 & T_1 <: T \\ \Gamma; \bar{f} \vdash t_2 \colon T_2 \,!\, \mathrm{e}_2 & T_2 <: T\end{array}}{\Gamma; \bar{f} \vdash \texttt{try}\ t_1\ \texttt{catch}\ (\bar{p})\ t_2 \colon T \,!\, \mathbf{eff}(\texttt{catch}\ (\bar{p}), \mathrm{e}_1, \mathrm{e}_2)}$$

then $\mathbf{eff}(\texttt{catch}\ (\bar{p}), \mathrm{e}_1, \mathrm{e}_2) = \texttt{throws}\ ((\bar{q} \backslash \bar{p}) \cup \bar{s})$ where $\mathrm{e}_1 = \texttt{throws}\ (\bar{q})$ and $\mathrm{e}_2 = \texttt{throws}\ (\bar{s})$, i.e., the **eff** function indicates that the possibly thrown exceptions are the exceptions $\bar{s}$ of e_2, and the exceptions $\bar{q}$ of e_1 minus the exceptions $\bar{p}$ captured by the `catch` expression.

To support subeffecting, i.e., ordering between effects, special subtyping rules are introduced. The rules are standard except when comparing effects. When comparing effects two components are taken into consideration: a concrete effect Ξ and a list of relative effects $\overline{x}$.

When comparing $(\Xi_1, \overline{x_1}) \preceq (\Xi_2, \overline{x_2})$, the effect Ξ_1 has to be a sub-effect of the effect Ξ_2 and each relative effect $x \in \overline{x}$ has to conform $x \preceq (\Xi_2, \overline{x_2})$. Either x exists as a relative effect in $\overline{x_2}$ or the expansion of the relative effect x conforms to $(\Xi_2, \overline{x_2})$.

2.6 Summary

In this chapter we have introduced several concepts required to understand our thesis. Section 2.1 introduced basic notions about effects and provided some examples of them. Section 2.2 introduced effect systems, which help programmers to control effects.

Section 2.2 introduced generic effect systems. We have shown that in general effect systems only allow programmers to work with one effect domain. Generic effect systems allow working with multiple domains at the same time. This system work as an effect privilege checking system, verifying if an expression has authorization to produce effectful operations.

Section 2.3 introduced gradual typing. We have shown how the conservative nature of static typing can be addressed by using gradual typing: mixing static and dynamic typing, getting the best of both worlds.

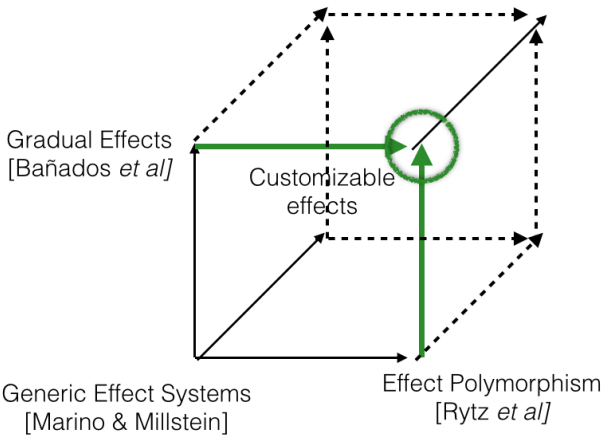
Section 2.4 introduced gradual effect systems, a generic effect system that works with gradual effects. The gradual effect system also works as an effect privilege checking system, but adding the unknown effect and new relations such as consistency between effect privileges. We have shown some type rules, also briefly showing how the insertion of cast is produced.

Section 2.5 introduced a practical effect system, an effect system that characterizes for the introduction of effect polymorphism, a high-order pattern for effect. Effect polymorphism allows expressing the effects of a function relative to its arguments. We also have shown how sub-effect is supported, to allow expressing ordering between effects.

In the following chapter we present a combined gradual effects from Bañados *et al* with effect polymorphism and sub-effecting from Rytz *et al*.

Chapter 3

A Combined Effect System



Now that we have the background about effects required to understand next chapters, we can combine two axis of the Effect Cube: gradual effects and effect polymorphism. We also include sub-effecting as it is contained in effect polymorphism. For the rest of the thesis we will refer with "TGE" to the Theory of Gradual Effects presented by Bañados *et al* and with "PES" to the Practical Effect System presented by Rytz *et al*.

We first start by explaining the need to combine both systems. Second, we formalize a combined effect system as an extension of the effect system introduced in TGE. We show how the type rules are extended along with its meta-functions. Third we explain the extension done to the translation rules to transform the source language into the internal language (casts insertions). We finalize the chapter by explaining the extension to the dynamic semantics.

3.1 The Problem

As previously mentioned, both PES and TGE are generic effect systems with outstanding

strengths. In particular, TGE proposed a **gradual** effect system providing programmers the flexibility to transition between a static and a dynamic effect system which allows to gradually adopt an effect system into an existing project. The effect system in PES introduced lightweight **effect polymorphism**, augmenting the expressiveness over effects. Also, PES support lattices with sub-effecting or ordering between effects, contrary to TGE which expresses effects as unordered sets.

We propose an effect system that focuses on being practical. To design a "practical" effect system, the effect system must have the strengths of PES and TGE, therefore gradual effects, effect polymorphism and sub-effecting must be included in the effect system. We start this work by extending the TGE effect system, adding effect polymorphism and sub-effecting.

This is based on a checking system

3.2 Formalization

The effect system supports gradual effects, effect polymorphism and sub-effecting. The formalization is based on the effect system defined in TGE, therefore, as mentioned in Section 2.4, the practical effect system works by checking effects where effects are considered as "privileges": an authorization to produce a particular effect. It is also important to mention that the source language, in contrast to TGE, is fully effect-annotated and consequently the signature of function definitions or function abstractions have their latent effects annotated. This last point is crucial for having a deterministic effect system as will be explained later.

Note about soundness of the system. It is important to mention that soundness of the combined effect system is proven in a separate technical report, to be published soon.

3.2.1 Annotations

When an effect annotation is referenced we will append `@` to the name. In order to illustrate examples in the following sections, suppose we are working with the domain of references with three effects annotations: `@write`, `@alloc` and `@read`. Let us call `@pure` the absence of effect or the bottom effect. `@unknown` is also used to denote the '¿' effect. To annotate a function abstraction that is effect-polymorphic on an argument `x` we will use `@pure(x)` as introduced in PES.

Suppose a function abstraction `f` typed $(\texttt{Int} \longrightarrow \texttt{Int}) \xrightarrow[\{x\}]{\{¿\}} \texttt{Int}$. The *latent* effect, or the effect that may happen when `f` is applied, is unknown ¿. `f` is also polymorphic on its argument x.

$$\phi \in \mathbf{Priv}, \quad \xi \in \mathbf{CPriv} = \mathbf{Priv} \cup \{¿\}$$
$$\Phi \in \mathbf{PrivSet} = \mathcal{P}(\mathbf{Priv}), \quad \Xi \in \mathbf{CPrivSet} = \mathcal{P}(\mathbf{CPriv})$$

v	::=	$\texttt{unit} \mid (\lambda x\colon T\ .\ \mathrm{e})^{T,\Xi,\overline{x}}$	Values
e	::=	$x \mid v \mid \mathrm{e}\ \mathrm{e}$	Terms
T	::=	$\texttt{Unit} \mid T \xrightarrow[\bar{x}]{\Xi} T$	Types
Γ	::=	$\emptyset \mid \Gamma, x : T$	Parameter context
$\bar{x}$	::=	$\emptyset \mid \bar{x}, x$	Polymorphic context

Figure 3.1: Language syntax

3.2.2 Consistent privilege set

A set of static effects or privileges Φ, represents the effect that are produced or are allowed during the evaluation of an expression. The elements of Φ are written ϕ. For example Φ can be $\{\texttt{@write}, \texttt{@alloc}\}$ to represent the effects `@write` and `@alloc`.

Similar to Φ, a "**consistent** privilege set", also called "privilege set" Ξ , is a privilege set annotations ξ. Notice that ξ can also be the unknown effect. For example Ξ can be $\{\texttt{@write}, \texttt{@alloc}, \texttt{@unknown}\}$, meaning that Ξ represents the effects `@write`, `@alloc` and the `@unknown` or ¿ effect.

3.3 Language Syntax

The syntax of the effect system is presented in Figure 3.1. As in TGE, the language is parameterized on some finite set of privileges **Priv** for a given effect domain. Subeffecting is a partial order on effect privileges, denoted $\phi_1 <: \phi_2$. A consistent privilege, in **CPriv**, can additionally be the unknown privilege ¿. A consistent privilege set Ξ is an element of the power set of **CPriv**, *i.e.* a set of privileges that can include ¿. The privilege set Ξ on a function declares their *latent* effect, the effect that may occur when the function is applied. A value can be `unit` or a function definition. $(\lambda x\colon T\ .\ \mathrm{e})^{T_2,\Xi,\overline{x}}$ represents a fully effect-annotated function abstraction, where T is the type of the function argument x, T_2 the return type, Ξ is the latent (consistent) privilege set or the privilege set authorised by the body of the function, $\overline{x}$ the relative effects and e is the body of the function. To illustrate a fully effect-annotated lambda let us consider a function typed $\texttt{Int} \xrightarrow[x]{\{\texttt{@write}\}} \texttt{Int}$ being effect-polymorphic on its argument x. The fully effect-annotated function in our system is denoted:

$$(\lambda x\colon \texttt{Int}\ .\ ...)^{\texttt{Int},\{\texttt{@write}\},x}$$

A term e can be a variable x, a value v or an application e e. The types can be `Unit` or a function type $T \xrightarrow[\bar{x}]{\Xi} T$ where Ξ is the latent privilege set and $\bar{x}$ the name of the parameters

that the function is being effect polymorphic. $\bar{x}$ is the set of variables that the context is being effect polymorphic called *relative effects*.

3.3.1 Typing rules

$\boxed{\Xi; \Gamma; \overline{x} \vdash \mathrm{e} : T}$

$$\text{C-Var}\frac{\Gamma(x) = T}{\Xi; \Gamma; \bar{x} \vdash x : T}$$

$$\text{C-Fn-An}\frac{\Xi_1; \Gamma, x : T_1; \overline{x} \vdash \mathrm{e} : T_2 \qquad T_2 \lesssim : T}{\Xi; \Gamma; \overline{x} \vdash (\lambda x : T_1 \ . \ \mathrm{e})^{T, \Xi_1, \overline{x}} : T_1 \xrightarrow[\overline{x}]{\Xi_1} T}$$

$$\text{C-App-Param}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi); \Gamma; \overline{x} \vdash f : T_1 \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_2 : T_2 \\ f \in \overline{x} \qquad T_2 \lesssim : T_1 \qquad \widetilde{\mathbf{check_{APP}}}(\Xi)\end{array}}{\Xi; \Gamma; \overline{x} \vdash f \ \mathrm{e}_2 : T}$$

$$\text{C-App}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_1 : T_1 \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_2 : T_2 \\ \Xi_1{}' = \Xi_1 \cup (\cup_{f \in (\overline{y} \setminus \overline{x})} latent((\Gamma, x : T_2)(f))) \qquad \Xi_1{}' \sqsubseteq : \Xi \qquad T_2 \lesssim : T_1 \\ \widetilde{\mathbf{check_{APP}}}(\Xi)\end{array}}{\Xi; \Gamma; \overline{x} \vdash \mathrm{e}_1 \ \mathrm{e}_2 : T_3}$$

$$\text{C-Eff}\frac{\Xi; \Gamma; \overline{x} \vdash \mathrm{e} : T \ ! \ \Xi \qquad \Xi \sqsubseteq : \Xi_1}{\Xi; \Gamma; \overline{x} \vdash (\mathrm{e} :: \Xi_1) : T}$$

Figure 3.2: Type system for the combined effect system

The type system is presented in the Figure 3.2. The type judgement $\Xi; \Gamma; \overline{x} \vdash \mathrm{e} : T$ is similar to the one presented in TGE adding to the judgement a set $\overline{x}$ that tracks the function parameters on which the expression e is being polymorphic.

The [C-Fn-An] rule is similar to the function abstraction defined in TGE except for the privilege set associated with the body of the function. [C-Fn-An] associates the privilege set annotated on the function definition with the body of the function, contrary to the original rule, that must deduce some privilege set.

As shown in the type rules, in order to support sub-effecting in the system, we first lift consistent containment defined in TGE to *consistent subcontainment*.

Definition 1 (Consistent Subcontainment)**.** Ξ_1 *is* consistently subcontained *in* Ξ_2*, notation* $\Xi_1 \sqsubseteq : \Xi_2$ *if and only if* $\Phi_1 \subseteq : \Phi_2$ *for* some $\Phi_1 \in \gamma(\Xi_1)$ *and* $\Phi_2 \in \gamma(\Xi_2)$.

The definition of subcontainment $\subseteq:$ extends the concept of containment to support sub-effecting.

Definition 2 (Subcontainment)**.** Φ_1 *is* subcontained *in* Φ_2*, notation* $\Phi_1 \sqsubseteq_{\approx}: \Phi_2$ *if and only if* $\forall \phi_1 \in \Phi_1, \phi_1 \in \Phi_2$ *or* $\exists \phi_2 \in \Phi_2$ *where* $\phi_1 <: \phi_2$

Consistent subtyping $\lesssim:$ is defined by replacing the containment rule defined in the subtyping rules in PES with consistent subcontainment. The subtyping rules takes into consideration relative effects. To illustrate a simplified version of subtyping, let us consider $T_1{}' \xrightarrow[\overline{x}']{\Xi_1{}'} T_2{}' \lesssim: T_1 \xrightarrow[\overline{x}]{\Xi_1} T_2$. The privilege sets $\Xi_1{}'$ and Ξ_1 are compared using consistent subcontainment. Every relative effect x' of $\overline{x}'$ either is contained in $\overline{x}$, or the expansion of effects of x' (extracted from the type of x') is contained in $\overline{x}$ recursively.

[C-App-Param] is a new rule when applying a function which is the parameter of an enclosing effect-polymorphic function. The difference between [C-App-Param] and [C-App] is very subtle: the typing rule [C-App-Param] does not need to check if the latent effects of the function being applied are consistent subcontained on the set of privileges of the enclosing application, i.e., $\Xi_1{}' \sqsubseteq_{\approx}: \Xi$. The reason is that in [C-App-Param] the application is being polymorphic on the applied function f, and by definition of effect polymorphism, the application is "allowed" to produce any effect that f may produce. The checking for the privileges over the invocation of f is done when f is passed as an argument to the invoking function that has relative effect f, and therefore the [C-App] rule applies.

The [C-App] rule is similar to the original application rule found in TGE, save for the expansion of relative effects. As e_1 may be effect-polymorphic, the latent effect of e_1 must be calculated according to its argument e_2. The latent effects of e_1 consist of the concrete effect Ξ_1 and the latent effect of the functions $f \in \overline{y}$ which are part of the relative effect of the function type.

Finally, the rule for effect ascription [C-Eff] is only altered to use consistent subcontainment instead of consistent containment.

3.3.2 Extension of the Translation Rules

The source language presented in section 3.3.1 supports unknown annotations, therefore runtime checks must be introduced. To introduce runtime checks, just like in TGE, the source language is translated into an internal language.

The internal language makes runtime checks explicit. The rules are identical to the one presented in TGE except for the support for effect polymorphism and sub-effecting. In the internal language, effectful operations must have enough privileges to be performed, consistency checks are resolved statically or runtime privilege checks guarantees satisfaction before reaching the effectful operation.

We illustrate the most important modifications to the translation rules of the original system for applications in figure 3.3.

$\boxed{\Xi; \Gamma; \overline{x} \vdash \mathrm{e} \Rightarrow \mathrm{e}' : T}$

$$\text{CTn-App-Param}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi); \overline{x} \vdash f \Rightarrow f : T_1 \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' : T_2 \\ \mathrm{e}_1'' = \langle\langle T_2 \xrightarrow{\Xi} T \Leftarrow T_1 \xrightarrow{\emptyset} T\rangle\rangle(\lambda x : T_1 \,.\, f\ x)^{T_3, \Xi_1, \overline{y}} \\ f \in \overline{x} \qquad T_2 \lesssim : T_1 \qquad \widetilde{\mathbf{check}}_{\mathbf{APP}}(\Xi) \qquad \Phi = \Delta(\Xi)\end{array}}{\Xi; \Gamma; \overline{x} \vdash f\ \mathrm{e}_2 \Rightarrow \mathit{insert\text{-}has?}(\Phi, \mathrm{e}_1''\ \mathrm{e}_2') : T}$$

$$\text{CTn-App}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' : T_1 \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' : T_2 \\ \Xi_1' = \Xi_1 \cup (\cup_{f \in (\overline{y} \setminus \overline{x})} \mathit{latent}((\Gamma, x : T_2)(f))) \qquad \Xi_1' \sqsubseteq : \Xi \qquad T_2 \lesssim : T_1 \\ \mathrm{e}_1'' = \langle\langle T_2 \xrightarrow{\Xi} T \Leftarrow T_1 \xrightarrow{\Xi_1'} T\rangle\rangle \mathrm{e}_1' \\ \widetilde{\mathbf{check}}_{\mathbf{APP}}(\Xi) \qquad\qquad \Phi = \Delta(\Xi)\end{array}}{\Xi; \Gamma; \overline{x} \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \mathit{insert\text{-}has?}(\Phi, \mathrm{e}_1''\ \mathrm{e}_2') : T}$$

Figure 3.3: Translation of source program to the internal language for applications

Given that casts work on function abstractions and f is a variable, to introduce a cast in [CTn-App-Param], the variable f must be encapsulated in an anonymous function. It is incorrect not to encapsulate f in a lambda because the runtime checks must be done only when the actual function is being applied. The cast also changes the latent effect of f to "pure" to prevent dynamic checks in the effects of a variable in which we are begin effect polymorphic. For the rule [CTn-App], the relative effects of e_1' must be taken into consideration when inserting the cast. Once again, the latent effects of e_1' considered in the cast, consist of the concrete effect Ξ_1 and the latent effects of the functions $f \in \overline{y}$ which are part of the relative effect of the function type.

3.3.3 Extension of the Dynamic Semantics

Until now, the effect system supports gradual effect, effect polymorphism and sub-effecting. We present now an example to show the necessity to modify the dynamic semantics to support effect-polymorphic casts.

Let us consider a high-order function `receiveMap` typed as:

$$\texttt{receiveMap} : (m : \texttt{List[Int]} \xrightarrow{\bot} (f : \texttt{Int} \xrightarrow{\top} \texttt{Int}) \xrightarrow[\{f\}]{\bot} \texttt{List[Int]}) \xrightarrow{\top} \texttt{Int}$$

Let us suppose that the body of `receiveMap` applies its argument with the identity function:

```
def receiveMap(m: ...) ... =  {
    m(List(), ((x: Int) => x))
    10
}
```

`receiveMap` takes a pure `map`-like function as argument and returns an `Int`. We are declaring that the argument m must be pure except for the effects that the function f may produce. Let us consider a function called `badMap` typed as

$$\texttt{badMap}\colon \texttt{List[Int]} \xrightarrow{\bot} (\texttt{Int} \xrightarrow{\top} \texttt{Int}) \xrightarrow{\{¿\}} \texttt{List[Int]}$$

If the implementation of `badMap` is

```
def badMap(l: List[Int], f: Int => Int): List[Int] @unknown = {
    write() // <- produces the @write effect!!
    l
}
```

When `receiveMap(badMap)` is applied with $\top$ as the initial set of privileges, a cast will be inserted:

$$\langle \texttt{List[Int]} \xrightarrow{\bot} (\texttt{Int} \xrightarrow{\top} \texttt{Int}) \xrightarrow{\{¿\}} \texttt{List[Int]} \xrightarrow{\top} \texttt{Int} \Leftarrow (m : \texttt{List[Int]} \xrightarrow{\bot} (f : \texttt{Int} \xrightarrow{\top} \texttt{Int}) \xrightarrow[\{f\}]{\bot} \texttt{List[Int]}) \xrightarrow{\top} \texttt{Int} \rangle$$

The cast will insert a `restrict`($\bot$) at runtime, throwing a runtime error inside the body of `badMap` because `write()` will not have authorization to be applied. A problem arise when the body of `badMap` applies the function `f`. Let us suppose the implementation of `badMap` as:

```
def badMap(l: List[Int], f: Int => Int): List[Int] @unknown = {
    f(10) // <- produces Top!!
    l
}
```

As `badMap` is not polymorphic in its argument `f` and its annotated with the effect privilege $\{¿\}$ a runtime check will be inserted before the execution of `f` asking for permission to produce $\top$. As the cast inserted by the application `receiveMap(badMap)` will restrict the set of privileges to $\bot$, the invocation of `f` will throw a runtime error. The problem is generated because when a function with unknown effects is cast into a function that is effect polymorphic, the information about relative effects is not dynamically propagated to the body of the unknown function. We need to maintain and update a dynamic environment of the parameters on which the expressions are being polymorphic.

Let us remember the rule [CTn-App-Param] where we transform the application of a function which is the parameter of an enclosing effect-polymorphic function "$f\ \mathrm{e}_2 \Rightarrow$ *insert-has?*(Φ, $f'\ \mathrm{e}_2{}'$)". We insert a `has` operator if the set Φ is not empty. We need to prevent the execution of the `has` operator if f is dynamically the parameter of an enclosing effect-polymorphic function. Nevertheless, at runtime when the actual application is performed, the variable is going to be substituted by a lambda losing the information about the variable name. To address that issue, we extend the `has` operator to "remember" the variable name being applied, redefining the metafunction *insert-has?* as follows:

$$\mathit{insert\text{-}has?}(\Phi, \mathrm{e}) = \begin{cases} \mathrm{e} & \texttt{if } \Phi = \emptyset \\ \texttt{has}_x\ \Phi\ \mathrm{e} & \texttt{if } \mathrm{e} = x\ \mathrm{e}_2 \wedge \Phi \neq \emptyset \\ \texttt{has}\ \Phi\ \mathrm{e} & \texttt{otherwise} \end{cases}$$

The dynamic semantics use an extended judgement of the form $\Phi; \Pi \vdash \mathrm{e} \to \mathrm{e}'$ adding a new environment Π to dynamically keep track of the relative effects. The cast rule is also extended as follows:

$$\langle (x_2 : T_{21}) \xrightarrow[\overline{x_2}]{\Xi_2} T_{22} \Leftarrow (x_1 : T_{11}) \xrightarrow[\overline{x_1}]{\Xi_1} T_{12} \rangle (\lambda x\colon T_{11}\ .\ \mathrm{e}) \to$$
$$(\lambda x\colon T_{21}\ .\ \langle T_{22} \Leftarrow T_{12} \rangle \texttt{restrict}\ \Xi_2\ \texttt{has}\ (|\Xi_1| \backslash |\Xi_2|))\ \texttt{releff}\ ([x_1/x_2]\overline{x_2} \backslash \overline{x_1})[(\langle T_{11} \Leftarrow T_{21} \rangle x)/x]\mathrm{e})$$

The `releff` expression adjusts the relative effects available in the dynamic extent of the current subexpression. When casting from the relative effects $\overline{x_1}$ to $\overline{x_2}$, `releff` adjusts the dynamic set of relative effects to the relative effects that are part of $\overline{x_2}$ and not part of $\overline{x_1}$. The name of x_2 must be also renamed to x_1 to correctly match the operator $\texttt{has}_{x_1}$ in the body of the casted function.

The dynamic semantics are extended adding four new rules as we shown in figure 3.4:

$$\text{E-Releff}\ \frac{\Phi; \Pi \cup \overline{x} \vdash \mathrm{e} \to \mathrm{e}'}{\Phi; \Pi \vdash \texttt{releff}\ \overline{x}\ \mathrm{e} \to \mathrm{e}'} \qquad \text{E-Has-Param-T}\ \frac{\Phi' \not\subseteq \Phi \quad x \in \Pi \quad \Phi; \Pi \vdash \mathrm{e} \to \mathrm{e}'}{\Phi; \Pi \vdash \texttt{has}_x\ \Phi'\ \mathrm{e} \to \texttt{has}_x\ \Phi'\ \mathrm{e}'}$$

$$\text{E-Has-Param-V}\ \frac{}{\Phi; \Pi \vdash \texttt{has}_x\ \Phi'\ v \to v} \qquad \text{E-Has-Param-F}\ \frac{\Phi' \not\subseteq \Phi \quad x \notin \Pi}{\Phi; \Pi \vdash \texttt{has}_x\ \Phi'\ \mathrm{e} \to \texttt{Error}}$$

Figure 3.4: Extension to the dynamic semantics

[E-Releff] just adjusts the dynamic set of relative effects. [E-Has-Param-T] allows a $\texttt{has}_x$ expression to be reduced when the privileges are not sufficient and x is a member of the set of relative effects. The [E-Has-Param-F] rule illustrate when $\texttt{has}_x$ does not hold and x is not a member of the set of relative effects, which will produce a runtime error.

3.4 Summary

In this chapter we have combined two axis of the Effect Cube: gradual effects and effect polymorphism. We have also added sub-effects to the combined effect system as it is part of effect polymorphism and allows us to build complex lattices. First section 3.1 we have shown the initial motivation about wanting a practical effect system that required both systems to be combined.

In section 3.2 we presented a formalization of the combined effect system, presenting the language syntax in section 3.3 and the type rules in section 3.3.1. The formalization of the combined effect system extends the effect system presented in TGE, with support for effect polymorphism and sub-effecting, therefore the combined effect system also works as an effect privilege checking. We have shown how some meta-functions were modified and also we defined new meta-functions required for the system.

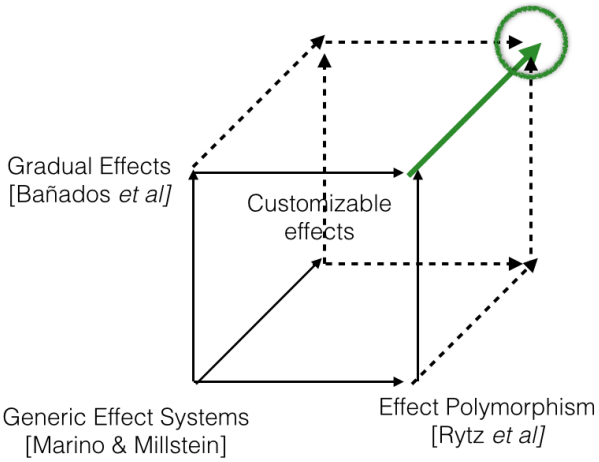
In section 3.3.2 we have explained the modifications done to the translation rules needed to modify the source code into the internal code. We have presented the type rules for function application to show how casts must be inserted when working with effect-polymorphism.

In section 3.3.3 we have shown with an example why the dynamic semantics must be extended too. The example shows that when casting an unknown function into a effect polymorphic one, the system needs to dynamically remember which variables are parameters of an enclosing effect-polymorphic function. We presented the modifications done to the meta-function *insert-has?*, then we shown the modifications done to casts and we finalize by showing the extension done to the dynamic semantics.

In the next chapter we show why a customizable effect system is necessary. We combine the last axis about customizable effects into our combined effect system, obtaining an effect system that combines gradual effects, effect polymorphism, sub-effecting and customizable effects.

Chapter 4

A Customizable Effect System



In the previous chapter we described an effect system that combines gradual effects, effect polymorphism and sub-effecting. In this chapter we extend that system with customizable effects.

In most cases, effect domains lack locality and therefore can be too strict for programmers. The programmer may want to specify where the effects must be produced and to do so, does not need to have expertice in effect system and in its implementation. We present a domain specific language to aid programmers in easily and creating effect disciplines by extending the combined effect system.

4.1 The Problem

4.1.1 Global scope of effect domains

The scope of existing effect domains is usually global within programming languages, i.e., effect domains lack locality and (un)pluggability. The programmer cannot easily customize

or tailor the effect domain according to his/her needs. Let us illustrate this point by quoting a user in the scala-users google groups:

> "I was wondering if there is a more or less precise definition of side effect. I have seen a few, more or less equivalent, but none can explain why I would consider *every* I/O operation as a side effect. Or, if I want to consider every I/O operation as a side effect, why I would consider *every* side effect when writing code. Taking everything as a potential danger looks a bit too much to me. ..."

and Wikipedia about referential transparency:

> "...Also, some impure functions can be included in the expression if their values are discarded and their side effects are insignificant."

Let us consider the I/O domain. Printing to screen produces side-effects, though they may not be of interest for the programmer to track. Suppose that the programmer wants to track only when an Integer is printed, or in other words:

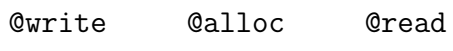
```
app print(Int) => @io
```

"Calling the function `print` passing an argument of type `Int` produces the effect `@io`". To express this kind of behavior an existing I/O domain must be modified or a new discipline must be created.

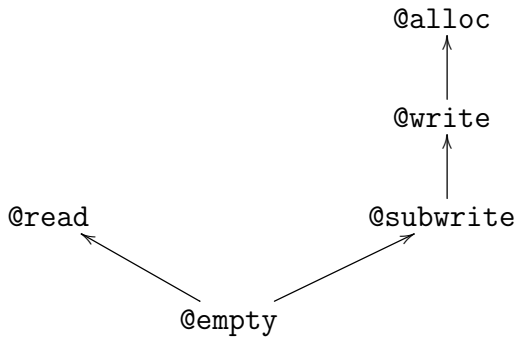
4.1.2 Complex lattices

Also, in order to design a generic effect system for multiple domains, we need to define an abstract representation for effects for every domain. The abstract representation of effects is different depending on the structure or ordering between effects. This relation between effects is called the *lattice* of the domain.

Suppose the reference domain with three mutable state effects: `@write`, `@read`, `@alloc`. The lattice of the reference domain can be graphically represented as:



No ordering between the effects `@write`, `@alloc` and `@read` exists. But an effect lattice can feature sub-effecting, i.e., ordering between effects. For instance, consider an alternative version of the reference domain called ARD for future reference:



The lattice introduces two new effects: `@empty` and `@subwrite`. Subeffecting is represented by the arrows in the figure, indicating that `@empty` <: `@subwrite` <: `@write` <: `@alloc` and that `@empty` <: `@read`. For example, given the set of initial privileges {`@alloc`}, an expression can produce `@alloc` or more precise effects: `@write`, `@subwrite`. Given that there is no relation between `@alloc` and `@read`, the same expression is not authorized to produce `@read`. The effect `@empty` is defined to represent the absence of effect over the domain.

Effect lattices can represent more complex and larger relations between effects. Sometimes, to formalize effect domains and their lattices, the type system must be modified or extended. For example, given a certain domain, the effect system described in PES must define the representation of every effect of the domain. The lattice must be able to express the sub-effect relations, how to join effects, which is the top and bottom effect. The function *eff* which infer effects must also be defined. As in PES, the effect system described in TGE have to be expanded to support a new domain, defining the **adjust** and **check** rules according to that domain. Effect lattices that feature ordering between effects are harder to be represented in TGE because the effect system expresses lattices as *sets* that form a lattice.

4.2 A practical language for a practical effect system

We propose a domain-specific language (DSL) to aid programmers in easily defining effect disciplines and to customize their effects, allowing programmers to work with application-specific effects. It also allows programmers to express complex lattices in a simple language syntax, and to define how and when effects must be tracked. An effect discipline is composed of an effect domain, its lattice, and the information of how and where the effects are produced.

Compiled libraries that do not have information about effects are considered to produce the larger set of effect possible: top. By using the DSL, the user can specify the latent effect of compiled functions, overriding the default value assigned by the effect system. This is useful when the user knows that the compiled or unknown function does not produce effects over the working effect disciplines. This method may be the best given that it is simpler than re-compiling the compiled external library (and only if the source code is available)

For example, the effects and the lattice of ARD can be represented as:

```
privileges:
    @empty
    @read
    @subwrite
    @write
    @alloc

lattice:
    top: @read @alloc
    bottom: @empty
    @empty <: @read
    @empty <: @subwrite
    @subwrite <: @write
    @write <: @alloc
```

Remember that top represent the maximum privilege set that may be produced by an expression in the domain. Top is defined as the union of the largest effects: `@read` and `@alloc`.

The DSL allows to declare when an effect is produced or what effects are authorized for a given expression. Declaring what effects are produced for a given expression introduces effects on a program without altering its source code. For instance, to express that a function named `insertDB` produces the effect `@write` when applied, one can declare it in the DSL like:

```
app insertDB() => set @write
```

The expression is declaring that the latent effect of any function named `insertDB` is {`@write` }, without the need to annotate it on the source code.

Programmers can use the DSL to specify certain constraints over function types. For instance, to allow the body of a function named `canWrite` typed `Int`⟶`Int` to produce the effects `@write` and `@subwrite` when applied one can declare it like:

```
def canWrite(T =:= Int): Int => set @write
```

Restricting the type of the first argument of the function `canWrite` to be equal to `Int`.

4.3 Syntax of the practical language

The DSL defines an effect discipline and its lattice by using the syntax defined in Figure 4.1. The non-terminal *typePattern* is a plain type name, and *namePattern* can either be a plain method name, the wildcard * (indicating all names), or an identifier with embedded `*` and `.` wildcards.

The DSL also allows declaring when an effect is produced or what effects are authorized for a given expression using *external effect specifications* which attach *updates* to *pointcuts*

privilegeDef	::=	`@unknown` | `@`*effectName typeParamP*	Effect privilege Definition
privilege	::=	`@unknown` | `@`*effectName typeParam*	Effect privilege
typeParamP	::=	`[`*variance type*, ...`]` | •	Type param with variance
typeParam	::=	`[`*type*, ...`]` | •	Type param
variance	::=	• | `+` | `-`	Type variance
type	::=	*typePattern comp* | *typePattern*	Type
comp	::=	`<:` *typePattern* | `:>` *typePattern* | `=:=` *typePattern*	Type constraint
ees	::=	*pointcut* ⇒ *update* ...	External effect specification
pointcut	::=	*selector signature*	Pointcut
selector	::=	`app` | `def`	Selector
signature	::=	*namePattern typeParam args*	Function signature
args	::=	• | `(`*type*, ...`)` | `()`	Function argument types
update	::=	*operation privilege* ...	Effect update
operation	::=	`add` | `set` | `remove`	Operation

Figure 4.1: Language syntax for the DSL

to update effects on certain part of the code. An effect update is composed of a an operation and a privilege set. Pointcuts are collections of join points. Join points are well defined locations within the source code where an action over effects is wanted. Join points can be a function definition an application or some other point in the execution of the program.

Effect disciplines are created using the DSL by defining three components: the effect privileges, the lattice, and the external effect specifications.

4.3.1 Effect privileges

First, the effects that compose the domain must be defined. The privileges are defined as a list of effect privilege definitions (*privilegeDef*). For instance, the list of privileges that define the reference domain previously used on the examples can be defined as:

```
privileges:
    @write
    @read
    @alloc
    @empty
```

Where `@empty` represents the absence of effects. When referring a pure function we use the annotation `@pure`, that applies for every effect domain. To specify the absence of effect on this particular effect domain, it is necessary to define an extra effect that represents the

bottom element of the lattice: `@empty`

Effects with type parameters

Effect privileges can be defined with type parameters. The variance of type parameters is invariant by default, but type parameters can be declared with variance annotations: `+` for covariant, `-` for contravariant. Type parameters can also be declared with type constraints. For instance,

```
privileges:
    @write
    @read[+T]
    @alloc[-T, V <: String]
    @empty
```

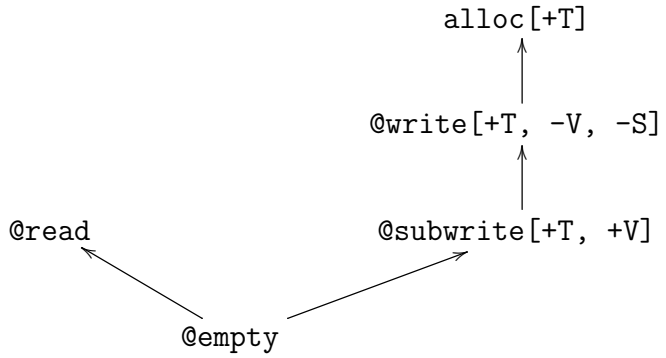
Defines the same three privileges of the previous example but `@read` now takes one type parameter that is covariant. The privilege `@alloc` receives two type parameters where the first one is contravariant. The second type parameter of `@alloc` is invariant and has a type constraint that indicates that `V` must be a subtype of String.

Two implicit subtyping relations are created for `@read` and `@alloc`. For `@read`: $\forall\ T_1 <: T_2$ *iff* `@read`$[T_1] <:$ `@read`$[T_2]$. For `@alloc`:

$$\forall\ T_1 <: T_2, V_1 =:= V_2, V_1 <: \texttt{String}\ \textit{iff}\ \texttt{@alloc}[T_2, V_1] <: \texttt{@alloc}[T_1, V_2]$$

4.3.2 Effect Lattice

Once the effect privileges are defined, the effect lattice must be defined where top and bottom are instantiated using the effect privileges previously defined. Additionally, sub-effect relations can be defined by a list of binary relations *typePattern comp*. Consider the following lattice:



The privilege section can be defined as:

```
privileges:
    @subwrite[+T,+V]
    @write[+T, -V, -S]
    @read
    @alloc[+T]
    @empty
```

The lattice section must define the top and bottom element, and represent the subtyping relation between effects:

```
lattice:
    top: @read @alloc[Any]
    bottom: @empty
    @subwrite[V, T] <: @write[A, T, C]
    @write[A, B, C] <: @alloc[B]
```

For `top`, `@read` and `@alloc` are combined. Given that the type parameter of `@alloc` is covariant and it does not have type constraints, `@alloc` is instantiated with the biggest type parameter possible (to illustrate the examples we are using the root of the Scala hierarchy): `Any`.

For `bottom`, an auxiliary effect privilege is created called `@empty` to represent the absence of effects for that domain.

To relate types in subtyping relations the same types must be used in the lattice, for example in `@subwrite[V, T] <: @write[A, T, C]` we are relating the second type parameter of `@subwrite` with the second type parameter of `@write`. No other subtyping relation is defined for the rest of the type parameters, therefore holding the relation for every type. When two type parameters are related in a subtyping declaration, the relation will take into consideration the variance of each type parameter.

Notice that for the relation `@write[A, B, C] <: @alloc[B]` we are relating the second type parameter of `@write` with the first type parameter of `@alloc`. By transitivity the second parameter of `subwrite` is also related with the the first type parameter of `@alloc`.

The privilege section and the lattice section allows programmers to quickly define ready-to-use domains and their lattices. For example, using the last example lattice to express that the body of a function named `canRead` typed `Int`$\longrightarrow$`Int` is authorized to produce the effect `@read`, can be declared in a Scala like syntax as follows:

```
def canRead(x: Int): Int @read = {
   ...
}
```

Authorizing a privilege with type parameters and subeffecting is illustrated in the following example

```
def canWrite(x: Int): Int @write[Object, Object, Foo] = {
   ...
```

```
    produceSubwrite()  // produces the effect @subwrite[String, String]
    ...
}
```

The application `produceSubwrite` is allowed to produce the effect `@subwrite[String, String]` because the second argument of the effect `subwrite` "`String`" is a subtype of the second argument of the authorized privilege `write` "`Object`".

4.3.3 External effect specifications

External effect specifications aid programmers in quickly annotating effects in the source code without the need to modify the source code. For example, it can be used to declare not to track the effect of a certain effectful function. In other words, external effect specifications specify what effects must be associated when an expression matches a join point. A join point is composed of a selector and a signature.

The selector `def` will match with function definition expressions. The body of the function definition will be authorized to produce the effects declared in the list of effect updates. For example,

```
def views.html.foo.apply() => set @empty
```

will restrict the body of the function named `foo.apply` defined inside the package `views.html` to be pure or `@empty`. As no type pattern is defined it will match every function by name regardless of its arguments.

The selector `app` will match with application expressions. The effect updates (*update*) indicate what effect privileges are necessary when applying a function, for example.

```
app scala.slick.*.insert* => set @write @alloc
```

indicates that the privilege set {`@write @alloc`} is required in order to apply any function that starts with `insert` defined inside the package "scala.slick".

To specify type parameters on effects, they must be associated with a type of the signature of the function. For instance, the declaration

```
app bar[T <: String](V) => add @subwrite[V, T]
```

matches applications of functions named `bar` which have a type parameter, and that are instantiated with some type parameter subtype of `String`. The invocation of `bar` requires the effect privilege `@subwrite` instantiated with the type of the first argument and the type parameter of "bar".

4.4 Formalization

The effect system for the DSL extends the effect system described in the previous chapter synthesizing TGE and PES. The effect system overrides the privilege set in such a way to maintain a connection between the functions **adjust** and **check** of TGE with the *eff* function of PES. First, some questions must be answered, for example: what is the connection between "`set` Ξ" and **check** in an application? Modifying the privilege set of an application with operations such as "`set` Ξ" means two things: the effect produced by the application must be overridden to Ξ, i.e., the *eff* function must return Ξ in PES, and therefore we must check that we have authorization to produce Ξ, i.e., the **check** function must check for Ξ in the current set of authorized privileges.

Modifying the privilege set of a function definition is a little different. Function definitions do not produce effects, but when they are applied, "`set` Ξ" means two things: Ξ is the latent effect of the function definition, and therefore the body must be allowed to produce Ξ, i.e., the **adjust** function modifies the set of privileges of the body of the function to be Ξ.

4.4.1 Language syntax

The syntax of the effect system is presented in figure 4.2 as an extension to the syntax presented in figure 3.1. It is important to note that now lambdas carry a label to indicate the "name" of the lambda which will be used to match the function with pointcuts.

v	::=	$\texttt{unit} \mid (\lambda x\colon T\ .\ \mathrm{e})_l^{T,\Xi,\overline{x}}$	Values
...			
Σ	::=	$\emptyset \mid \Sigma, ees$	External effect specification set
ω	::=	$\bullet \mid \texttt{set} \mid \texttt{add} \mid \texttt{remove}$	Operator

Figure 4.2: Language syntax

A value can be `unit` or a labeled function definition. $(\lambda x\colon T\ .\ \mathrm{e})_l^{T_2,\Xi,\overline{x}}$ represents a fully effect-annotated function abstraction, where l is a label to represent its name.

To illustrate a fully effect-annotated lambda let us consider a function called `writeFun` typed $\texttt{Int} \xrightarrow[x]{\{\texttt{@write}\}} \texttt{Int}$ being effect-polymorphic on its argument x. The fully effect-annotated function called `writeFun`, in our system is denoted as:

$$(\lambda x\colon \texttt{Int}\ .\ ...)^{\texttt{Int},\{\texttt{@write}\},x}_{\texttt{writeFun}}$$

Σ stores the external effect specifications defined in the DSL. External effect specifications are used to match expressions and modify sets of effects depending on the expression. ω

represents an operation over a set of effect Ξ which can be add, remove or replace the privilege set. For example "`add {@read, @write}`" will add the effects `@read` and `@write` to the current privilege set.

4.4.2 Typing rules

$\boxed{\Sigma; \Xi; \Gamma; \overline{x} \vdash \mathrm{e} : T}$

$$\text{T-Var}\ \frac{\Gamma(x) = T}{\Sigma; \Xi; \Gamma; \bar{x} \vdash x : T}$$

$$\text{DT-Fn-An}\ \frac{\begin{array}{c}\omega\ \Xi_\Sigma = \mathbf{match}(\Sigma, \mathtt{def}, l, T_1 \longrightarrow T) \\ \Sigma; \mathbf{update}(\omega, \Xi_\Sigma, \Xi_1); \Gamma, x : T_1; \overline{x} \vdash \mathrm{e} : T_2 \qquad T_2 \lesssim : T\end{array}}{\Sigma; \Xi; \Gamma; \overline{x} \vdash (\lambda x : T_1\ .\ \mathrm{e})_l^{T, \Xi_1, \overline{x}} : T_1 \xrightarrow[\overline{x}]{\Xi_1} T}$$

$$\text{DT-App-Param}\ \frac{\begin{array}{c}\Sigma; \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi); \Gamma; \overline{x} \vdash f : T_1 \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \Sigma; \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_2 : T_2 \\ \omega\ \Xi_\Sigma = \mathbf{match}(\Sigma, \mathtt{app}, f, \Gamma(f)) \\ f \in \overline{x} \qquad T_2 \lesssim : T_1 \qquad \widetilde{\mathbf{\Sigma check}}_{\mathbf{APP}}(\Xi, \omega, \Xi_\Sigma)\end{array}}{\Sigma; \Xi; \Gamma; \overline{x} \vdash f\ \mathrm{e}_2 : T}$$

$$\text{DT-App}\ \frac{\begin{array}{c}\Sigma; \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_1 : T_1 \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \Sigma; \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_2 : T_2 \\ \omega\ \Xi_\Sigma = \mathbf{match}(\Sigma, \mathtt{app}, \mathrm{e}_1, T_1 \longrightarrow T_3) \\ \Xi_1{}' = \Xi_1 \cup (\cup_{f \in (\overline{y} \setminus \overline{x})} \mathit{latent}((\Gamma, x : T_2)(f))) \qquad \Xi_1{}' \sqsubseteq : \Xi \qquad T_2 \lesssim : T_1 \\ \widetilde{\mathbf{\Sigma check}}_{\mathbf{APP}}(\Xi, \omega, \Xi_\Sigma)\end{array}}{\Sigma; \Xi; \Gamma; \overline{x} \vdash \mathrm{e}_1\ \mathrm{e}_2 : T}$$

$$\text{DT-Eff}\ \frac{\Sigma; \Xi; \Gamma; \overline{x} \vdash \mathrm{e} : T\ !\ \Xi \qquad \Xi \sqsubseteq : \Xi_1}{\Sigma; \Xi; \Gamma; \overline{x} \vdash (\mathrm{e} :: \Xi_1) : T}$$

Figure 4.3: Type system for the customizable effect system

The type system is presented in the Figure 4.3. The type judgement $\Sigma; \Xi; \Gamma; \overline{x} \vdash \mathrm{e} : T$ is similar to the one presented in figure 3.2, adding a set of external effect specifications Σ into the judgement so the system can update the privilege set for every expression that matches some pointcut.

The [DT-Fn-An] rule associates a privilege set with the body of the function. That privilege set can be altered by the meta-function **update** only if there is a match on some join point of the DSL. The lambda can only match "`def`" join points by comparing the label and the types using the meta-function **match**.

Joint point	Matches...
`f`	functions called `f`
`f*`	functions which name starts with `f`
`*fun*`	functions which name contains `fun`
`scala.*.f`	functions called `f` which are defined inside any packages that is also contained in the package called `scala`
`f(T)`	functions called `f` that receives one or more arguments
`f(T): String`	functions called `f` that receives one or more arguments and returns a `String`
`f(T<:  String, V =:= Int)`	functions called `f` that receives two or more arguments The first argument must be a subtype of `String` The second argument must be of type `Int`

Table 4.1: How the meta-function **match** matches join points with function signatures

match matches a function signature of a function definition or a function application with pointcuts and returns an operation and a privilege set which will be used by **update** to update the privilege set. **match** receives as argument a set of external effect specifications Σ, a selector to indicate if its matching function applications (`app`) or function definitions (`def`), the name of the function and the type of that function. The meta-function matches pointcuts with function signatures as illustrated in table 4.1.

As mentioned the sets of effect privileges are "updated" by the function **update** which allows programmers to customize and tailor the effect system.

Definition 3 (Effect Privilege Override)**.**
Let **update** : $(\omega, \Xi', \Xi) \rightarrow \Xi$ *be defined as follows:*

$$\textbf{update}(\omega, \Xi', \Xi) = \begin{cases} \Xi' & \text{if } \omega' = \texttt{set} \\ \Xi \cup \Xi' & \text{if } \omega' = \texttt{add} \\ \Xi \backslash_{\Sigma} \Xi' & \text{if } \omega' = \texttt{remove} \\ \Xi & \text{if } \omega' = \bullet \end{cases}$$

The operator `set` replaces the privilege set to be Ξ'. The operator `add` represents extending the privilege set with Ξ'. The operation `remove` represents removing the effect Ξ' from the original privilege set Ξ. For function abstraction `set` represents an override of the effect annotations Ξ. Note that the **update** function only modifies the effects if the expression matches with a join point of the DSL. If there is no match then the operation $\bullet$ is returned and the original set of effect privilege is left intact.

Given that there is subsumption of effects, removing an effect Ξ' from the effect Ξ, or $\Xi \backslash_{\Sigma} \Xi'$, is defined as follows:

Definition 4 (Effect Removal)**.**

$$\Xi \backslash_{\Sigma} \Xi' = \{\Xi \backslash \xi \mid \forall \xi \in \Xi, \exists \xi' \in \Xi', \xi <: \xi'\}$$

The definition for Effect Removal $\Xi \backslash_\Sigma \Xi'$ only removes elements ξ from Ξ when there is an element ξ' in Ξ' that is a super type of ξ. Consider the lattice of effects `A` , `B` and `C` , with `@B` <: `@A` and `@C` <: `@A` . It is incorrect to say that $\{$`@A` $\}\backslash_\Sigma\{$`@B` $\} = \{$`@C` $\}$. According to the definition of Effect Removal, $\{$`@A` $\}\backslash_\Sigma\{$`@B` $\} = \{$`@A` $\}$. On the other hand, $\{$`@B` $\}\backslash_\Sigma\{$`@A` $\} = \{\emptyset\}$. The Effect Removal function removes all the effects from Ξ that are not a subtype of Ξ'. The reason for this behavior is that the effect `@A` is not necessarily the union of all its "sub-effects" `@B` and `@C` , i.e., $\{$`@A` $\} \neq \{$`@B` , `@C` $\}$. For example, given the IO effect domain and `@noIo` <: `@io` , it is **incorrect** to say that $\{$`@io` $\}\backslash_\Sigma\{$`@noIo` $\} = \{\emptyset\}$.

In the [DT-App-Param] and [DT-App] rules, the effect-checking is done by using the function $\widetilde{\mathbf{\Sigma check}}_{\mathbf{APP}}$. In case of a match of the expression on a "`app` join point" the function $\widetilde{\mathbf{\Sigma check}}_{\mathbf{APP}}$ alters the privilege set necessary to fulfill the requirements of the function $\widetilde{\mathbf{check}}_{\mathbf{APP}}$. By updating the necessary privilege set we can allow or restrict applications globally on the source code. For example, if the function $\widetilde{\mathbf{check}}_{\mathbf{APP}}$ checks the presence of the `@write`, updating the necessary set to `@pure` would accept every application on the code regardless of the initial privilege set.

The $\widetilde{\mathbf{\Sigma check}}_{\mathbf{C}}$ function is defined as follows:

$$\widetilde{\mathbf{\Sigma check}}_{\mathbf{C}}(\Xi, \omega, \Xi_\Sigma) = \begin{cases} \mathbf{update}(\omega, \Xi_\Sigma, \mathbf{needed}(\Xi)) \sqsubseteq_\sim \Xi & \text{if } \omega' \neq \bullet \\ \widetilde{\mathbf{check}}_{\mathbf{C}}(\Xi) & \text{if } \omega' = \bullet \end{cases}$$

The **needed** function calculates the minimal privilege set to safely pass the $\widetilde{\mathbf{check}}_{\mathbf{C}}$ function. The **needed** function is defined as follows:

$$\mathbf{needed}(\Xi) = \bigcup mins(\{\Phi \in \gamma(\Xi) \mid \mathbf{check}_{\mathbf{C}}(\Phi)\})$$

4.4.3 Extension to the transformation rules

The modifications to the transformation rules are shown in figure 4.4.3.

They are similar to the rules described in figure 3.3, but with slight modifications incorporating $\widetilde{\mathbf{\Sigma check}}_{\mathbf{C}}$ and **update**. One important change is that of changing the function Δ, which calculates the minimal set of additional set of privileges needed to safely pass the **check** function. Now a new function Δ^Σ is used which takes into consideration the overriding of effects done by the DSL by first calculating the needed privilege set. If there is a match on some pointcut, then it updates the needed set of effect using the function **update**. The Δ^Σ function is defined as follows:

$$\Delta^\Sigma(\Xi) = |\mathbf{update}(\omega, \Xi_\Sigma, \mathbf{needed}(\Xi))| \backslash |\Xi|$$

$$\text{DTn-App-Param}\ \frac{\begin{array}{c}\Sigma; \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi); \Gamma; \overline{x} \vdash f \Rightarrow f' \colon T_1 \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \Sigma; \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon T_2 \\ \omega\ \Xi_\Sigma = \mathbf{match}(\Sigma, \mathtt{app}, f, \Gamma(f)) \\ \mathrm{e}_1'' = \langle\langle T_2 \xrightarrow{\Xi} T \Leftarrow T_1 \xrightarrow{\emptyset} T\rangle\rangle(\lambda x \colon T_1\ .\ f'\ x)^{T_3, \Xi_1, \overline{y}}_{\bullet} \\ f \in \overline{x} \quad T_2 \lesssim \colon T_1 \quad \widetilde{\mathbf{\Sigma check_{APP}}}(\Xi, \omega, \Xi_\Sigma) \quad \Phi = \Delta^\Sigma(\Xi, \omega, \Xi_\Sigma)\end{array}}{\Sigma; \Xi; \Gamma; \overline{x} \vdash f\ \mathrm{e}_2 \Rightarrow \mathit{insert\text{-}has?}(\Phi, f'\ \mathrm{e}_2') \colon T}$$

$$\text{DTn-App}\ \frac{\begin{array}{c}\Sigma; \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon T_1 \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \Sigma; \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi); \Gamma; \overline{x} \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon T_2 \\ \omega\ \Xi_\Sigma = \mathbf{match}(\Sigma, \mathtt{app}, \mathrm{e}_1, T_1 \longrightarrow T_3) \\ \Xi_1' = \Xi_1 \cup (\cup_{f \in (\overline{y} \setminus \overline{x})} \mathit{latent}((\Gamma, x \colon T_2)(f))) \quad \Xi_1' \sqsubseteq \colon \Xi \quad T_2 \lesssim \colon T_1 \\ \mathrm{e}_1'' = \langle\langle T_2 \xrightarrow{\Xi} T \Leftarrow T_1 \xrightarrow{\Xi_1'} T\rangle\rangle \mathrm{e}_1' \\ \widetilde{\mathbf{\Sigma check_{APP}}}(\Xi, \omega, \Xi_\Sigma) \qquad \Phi = \Delta^\Sigma(\Xi, \omega, \Xi_\Sigma)\end{array}}{\Sigma; \Xi; \Gamma; \overline{x} \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \mathit{insert\text{-}has?}(\Phi, \mathrm{e}_1''\ \mathrm{e}_2') \colon T}$$

Figure 4.4: Transformation rules for the inclusion of external effect specifications

4.4.4 Examples

To illustrate the potentiality and flexibility of the DSL, let us consider two sets of domains. The first domain is the reference domain mentioned earlier with effects `@write`, `@read` and `@alloc`. The second domain is the I/O domain with effects `@io` and `@noIo` that represent top and bottom effects respectively. Let us consider a function called `canPrint` typed $\mathtt{Int} \xrightarrow{\{\mathtt{@io}\}} \mathtt{Int}$. Let us also consider a function `h` as part of an external compiled library, and consequently typed $\mathtt{Int} \xrightarrow{\{\top\}} \mathtt{Int}$ given that there is no information about effects.

$$(\lambda x \colon \mathtt{Int}\ .\ \mathtt{h}(x))^{\mathtt{Int}, \{\mathtt{@io}\}, x}_{\mathtt{canPrint}}$$

The type system will reject the application of `h` inside the body of `writeFun`. If the programmer knows that `h` produces the effect `@io`, an external effect specification can be added:

```
app h(Int): Int => set @io
```

This way, the expression will be accepted by the type system. Let us consider that the programmer realized that he made a mistake and notices that the function `h` also has the latent effect `@write`. The programmer can surely modify `writeFun` authorized privileges on the source code, but he can do it instead in the DSL by defining two external effect specifications:

```
app h(Int): Int => set @write @io
def writeFun(Int): Int => add @write
```

It is important to mention that it is incorrect to declare that the latent effect of `h` is ¿ or unknown. The reason for that is because the compiler can not introduce runtime checks into

the body of `h` given that it is an external compiled library and therefore the source code can not be altered.

If we add just one external effect specification: `def * => set @unknown` and add external effect specification on applications of effectful expressions that want to be tracked, we see how easily an existing code can adapt the effect system without the need to annotate every function either with unknown or with the actual effect privileges, This would result in a fully dynamic effect system with almost no modification to the code. From that point programmers can add effect annotations to gradually reduce the runtime checks.

4.5 Summary

In this chapter we have merged the three axes of the effect cube, combining gradual effects, effect polymorphism, sub-effecting and customizable effects. First, in section 4.1 we start explaining why a customizable effect system is necessary. We have shown that effect systems lack locality, are not easy to modify, and hard to represent complex lattices.

In section 4.3 we presented a domain-specific language that addresses the problems described in 4.1 by the introduction of external effect specifications. First, we start by describing the language syntax for the DSL. Second, in section 4.3.1 we explain with examples the component of the DSL where the effect privileges are defined. Third, in section 4.3.2 we explain with examples how lattices must be declared in the DSL. Finally we have shown in section in 4.3.3 how external effect specifications can be used.

In section 4.4 we presented a formalization of the customizable effect system, presenting the language syntax in section 4.4.1 and the type rules on section 4.4.2. The formalization of the combined effect system extends the effect system presented in the previous chapter, with support for updating of effects through external effect specifications. In section 4.4.3 we explain the extension to the transformation rules of the combined effect system, showing the transformation rules for function applications.

We finalized this chapter in section 4.4.4, by presenting some small examples of use of the customizable effect system. More sophisticated examples are presented in chapter 6.

The next chapter we presents the implementation for the customizable effect system, developed as Scala plugins.

Chapter 5

Implementation

In the previous chapter we presented the formalization of the customizable effect system that allows programmers to tailor their effect disciplines. This chapter explains how the customizable effect system is implemented into the Scala programming language.

This chapter starts by explaining in Section 5.1 how the practical effect system is implemented as a Bidirectional effect system and why a fully effect-annotated language is necessary. Section 5.2 describes the implementation in the Scala language as compiler plugins.

5.1 Bidirectional effect system

The Practical Effect System described in Chapter 4 is based on a fully effect-annotated source language which will be explained in this section.

Let us recall that the Practical Effect System is based on the effect system described in TGE. The major modification to the original rules in TGE, not taking into consideration effect polymorphism, effect-subtyping and the support for external effect specifications, is in the function abstraction rule.

The function abstraction rule defined in the source language of the work in TGE is non-deterministic. To illustrate it, we present the original rule:

$$\text{T-Fn}\frac{\Xi_1; \Gamma, x\colon T_1 \vdash \mathrm{e}\colon T_2}{\Xi; \Gamma \vdash (\lambda x\colon T_1\ .\ \mathrm{e})\colon T_1 \xrightarrow{\Xi_1} T_2}$$

A function definition does not produce side effects, but when applied it can produce some privilege set different than the privilege set associated with the function definition. Rather than using Ξ to type check the body of the function e, some set Ξ_1 is used. In other words, some privilege set Ξ_1 exists where the expression e is accepted by the type checker. The privilege set Ξ_1 must be deduced, which makes the system non-deterministic. For example,

let us consider the reference domain, a function h typed $\texttt{Int} \xrightarrow{\{\texttt{@write}\}} \texttt{Int}$, and the following function:

$$(\lambda x\colon \texttt{Int}\ .\ h(x))$$

The body of the lambda is authorized to produce the effect `@write` by associating the body with any of the set of privileges {`@write`}, {`@write`, `@alloc`}, {`@write`, `@read`, `@alloc`}, and as `@unknown` is consistent with `@write`, also {`@unknown`}, {`@unknown, @read`}, and so on.

The practical effect system is fully deterministic and work by "checking effects". To work with a purely "checking of effects" mode, it is necessary to know the latent effects beforehand, i.e., as several sets of effect privileges can be associated with the body of a function, first a sufficient set of privilege must be deduced. The non-determinist rule can be altered to be deterministic by using a *bidirectional* effect system. A bidirectional effect system, as the name describes, works in two different modes: checking and inference of effects. The system can switch between both modes any time according to the type rules. As explained earlier, "effect checking " works with effects as input of the type system, while "effect inference" works with effects as output of the system. By using a bidirectional system when type checking the body of the function, the mode of the effect system can change to "inference" to obtain a minimal privilege set Ξ_1.

Let us remember that the effect system of PES is all about effect inference. Given that in PES, for function abstraction, the system infers the maximum privilege set Ξ_{infer} that the body of the function may produce. In effect checking, $\Xi_1 = \Xi_{infer}$ can be used as an initial set of privileges.

To implement the bidirectional system, the system is divided in two steps. The first step is to translate the source language into a fully effect-annotated source language using effect inference. The second step is to use the effect checking system described in Section 4.4.2 on an already fully effect-annotated system. For instance, the last example with labeled functions, is transformed like this:

$$(\lambda x\colon \texttt{Int}\ .\ h(x))_{funName} \Rightarrow (\lambda x\colon \texttt{Int}\ .\ h(x))^{\texttt{Int},\{\texttt{@write}\},\emptyset}_{funName}$$

The bidirectional effect system is summarized in Figure 5.1

Figure 5.1: Components of the Bidirectional System

Translation to a fully effect-annotated source language The first step about the practical effect system is about inferring the set of effect of expressions on the source language. This is based on the type system defined in PES, adding the support for gradual effects and the support for the DSL. To support gradual effects, the effect annotations e are replaced by

$\boxed{\Delta; \Gamma; \overline{x} \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T \mathbin{!} \Xi}$

$$\text{Tn-Var}\frac{\Delta; \Gamma(x) = T}{\Delta; \Gamma; \overline{x} \vdash x \Rightarrow x \colon T \mathbin{!} \emptyset}$$

$$\text{Tn-Fn}\frac{\begin{array}{c}\Delta; \Gamma, x \colon T_1; \emptyset \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T \mathbin{!} \Xi_1 \\ \omega\ \Xi_\Delta = \mathbf{match}(\Delta, \mathtt{def}, l, T_1 \longrightarrow T) \\ \Xi' = \mathbf{update}(\omega, \Xi_\Delta, \Xi_1) \qquad \Xi_1 \sqsubseteq_{\sim} \colon \Xi'\end{array}}{\Delta; \Gamma; \overline{x} \vdash (\lambda x \colon T_1 \ .\ \mathrm{e})_l \Rightarrow (\lambda x \colon T_1 \ .\ \mathrm{e})_l^{T, \Xi', \emptyset} \colon T_1 \xrightarrow[\emptyset]{\Xi'} T \mathbin{!} \emptyset}$$

$$\text{Tn-Fn-An}\frac{\begin{array}{c}\Delta; \Gamma, \overline{x \colon T_x}; \overline{x} \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T_2 \mathbin{!} \Xi_1 \\ \omega\ \Xi_\Delta = \mathbf{match}(\Delta, \mathtt{def}, l, T_1 \longrightarrow T) \\ \Xi' = \mathbf{update}(\omega, \Xi_\Delta, \Xi) \qquad \Xi_1 \sqsubseteq_{\sim} \colon \Xi' \qquad T_2 \lesssim T\end{array}}{\Delta; \Gamma; \overline{x} \vdash (\lambda x \colon T_1 \ .\ \mathrm{e})_l^{T, \Xi, \overline{x}} \Rightarrow (\lambda x \colon T_1 \ .\ \mathrm{e})_l^{T, \Xi', \overline{x}} \colon T_1 \xrightarrow[\overline{x}]{\Xi'} T \mathbin{!} \emptyset}$$

$$\text{Tn-App-Param}\frac{\begin{array}{c}f \in \bar{f} \qquad \Delta; \Gamma; \overline{x} \vdash f \Rightarrow f \colon T_1 \xrightarrow[\overline{x}]{\Xi_1} T_3 \mathbin{!} \emptyset \qquad \Delta; \Gamma; \overline{x} \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon T_2 \mathbin{!} \Xi'' \qquad T_2 \lesssim T_1 \\ \omega\ \Xi_\Delta = \mathbf{match}(\Delta, \mathtt{call}, f, \Gamma(f)) \qquad\qquad \Xi = \mathit{eff}_{APP}(\emptyset, \Xi'', \emptyset)\end{array}}{\Delta; \Gamma; \overline{x} \vdash f(\mathrm{e}_2) \Rightarrow f(\mathrm{e}_2') \colon T_3 \mathbin{!} \mathbf{update}(\omega, \Xi_\Delta, \Xi)}$$

$$\text{Tn-App}\frac{\begin{array}{c}f \in \bar{f} \qquad \Delta; \Gamma; \overline{x} \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon T_1 \xrightarrow[\overline{x}]{\Xi_1} T_3 \mathbin{!} \Xi' \qquad \Delta; \Gamma; \overline{x} \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon T_2 \mathbin{!} \Xi'' \\ T_2 \lesssim T_1 \qquad\qquad \Xi_1 = \cup_{f \in (\overline{y} \setminus \overline{x})} \mathit{latent}((\Gamma, x \colon T_2)(f)) \\ \omega\ \Xi_\Delta = \mathbf{match}(\Delta, \mathtt{call}, \mathrm{e}_1, T_1 \longrightarrow T_3) \qquad \Xi = \mathit{eff}_{APP}(\Xi', \Xi'', \Xi_1 \cup \Xi_p)\end{array}}{\Delta; \Gamma; \overline{x} \vdash \mathrm{e}_1(\mathrm{e}_2) \Rightarrow \mathrm{e}_1'(\mathrm{e}_2') \colon T_3 \mathbin{!} \mathbf{update}(\omega, \Xi_\Delta, \Xi)}$$

$$\text{Tn-Eff}\frac{\Delta; \Gamma; \overline{x} \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T \mathbin{!} \Xi \qquad \Xi \sqsubseteq_{\sim} \colon \Xi_1}{\Delta; \Gamma; \overline{x} \vdash (\mathrm{e} :: \Xi_1) \Rightarrow (\mathrm{e}' :: \Xi_1) \colon T \mathbin{!} \Xi_1}$$

Figure 5.2: Translation rules from the source language into a fully annotated source language using the DSL.

a consistent set of effect annotations Ξ which may or may not contain the unknown effect. To recall some fundamental concept, to infer effects, the effect system uses the function *eff*. The function *eff* by default joins the different inferred effects by the sub-expressions but can be customized for the different type rules.

Because the modifications to the effect system defined in PES are standard and illustrated in the transformation rules, only the transformation rules are presented. Figure 5.2 presents the transformation rules from the source language into the fully effect-annotated language through effect inference.

The judgement $\Sigma; \Gamma; \overline{f} \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T \mathbin{!} \Xi$ means that the expression e translated into e' has type T and effect Ξ in the lexical environment Γ and Σ as the set of external effect specifications. Based on the judgement, the effect inferred for the expression e can be updated

according to the rules defined on the DSL.

The difference between [Tn-Fn] and [Tn-Fn-An] rules is that the first one is translating a function that is not effect-annotated, and [Tn-Fn-An] transform an effect-annotated function. The [Tn-Fn] and [Tn-Fn-An] rules illustrate how the latent effect Ξ_1 of function labeled l can be altered given the inferred effect Ξ_1. Note that the rules check if the inferred effect Ξ_1 is consistent contained in the updated privilege set Ξ'. In the [Tn-Fn-An] rule, the DSL update the privilege set annotated, so the privilege set used on the **update** function is the annotated privilege set, not the inferred effects on the expression that represents the body of the function abstraction.

The rules [D-App-Param] and [D-App] modify the set of inferred effects by the function *eff*. Note that the type used to match the expression on the DSL context is the type of the argument T_2 and not the type T_1 of the function being applied. As explained before, altering the inferred effect of an application is useful when the information on effects of the function being applied is not available. For example, using the the function `map` on Scala, one must use a pre compiled library "scala.collection". For the conservatism of the original type system, type checking the invocation of the function `map` will be the biggest effect possible for every domain of effects. This can be addressed by adding the external effect specification `app map => set @pure` altering the inferred effects from the invocation of `map` to be `@pure` .

5.2 Scala implementation

If the reader is more interested in conceptual concepts, he/she can skip this section as it is very technical. The implementation of the effect system can be obtained from `http://pleiad.cl/effscript`.

Why Scala? Scala was chosen not only because there is an existing implementation of an effect system that can be extended. Scala is a fast growing language, widely used in the industry, and has a very rich type system. It combines object oriented programming with functional programming. Most importantly, Scala is extensible through compiler plugins, not requiring the modification of the compiler itself. All this makes Scala a perfect basis to build this work upon.

The implementation of the Practical Effect system is developed as a compiler plugin for the Scala programming language. The plugin is based on the plugin developed by Rytz *et al* and is composed of two sub plugins to implement bidirectional checking: the effect inference plugin, and the effect checking plugin. The effect inference plugin is a modification of the one developed by Rytz *et al*, extended with support for gradual effects and the customizable effect system. As explained before, in relation to bidirectional checking, Scala has inference of effect, therefore there are cases where there is absence of effect annotations. The effect inference plugin is necessary to annotate function abstractions that do not have effect annotations. This information about effect inference is used by the effect checking plugin to check and adjust sets of effect privileges, also inserting runtime checks of effect wherever it may be

necessary.

The first plugin on effect inference uses the inference type system and the transformation type system described in the previous chapter. The implementation extends the typing system with support for classes, objects, lazy values, call by name parameters, etc. The second plugin on effect checking uses the type system described also in section 4.4.2.

The effect inference plugin is inserted in the "typer" compiler phase, overriding certain functions of that phase. The effect checker plugin is inserted **after** the "typer" compilation phase, using all the information inferred about effects.

5.2.1 Annotations as Effects

Effects are represented in the Scala programming language as annotations. An annotation has the symbol `@` as a prefix and is followed by a constructor of a class which must conform to the class `scala.Annotation`. Multiple annotations are annotated by separating them by a space.

In function signatures, the return type of the function can be declared with annotations to represent the latent effects of that function (or the effect that the body of the function is authorized to produce). For example, the signature of a function `f` typed $\texttt{Int} \xrightarrow{\texttt{\{@write\}}} \texttt{Int}$ can be written:

```
def f(x: Int): Int @write
```

A function with multiple effects `g` typed $Int \xrightarrow{\texttt{\{@write,@unknown\}}} Int$ can be declared:

```
def g(x: Int): Int @write @unknown
```

Also, effect annotations are used in effect ascription. For example, to adjust the privilege set for a function application `g(10)` to be `@unknown` one can use:

```
g(10) :: @unknown
```

As with function declarations, multiple effect annotations can be used on effect ascriptions.

The type of `g` can be observed in the Scala REPL as `Int => Int @write @unknown` .

5.2.2 Domains

To work with effects one must implement effect domains. Each effect domain is implemented by defining the annotations and a *lattice*. The annotations are implemented by defining the classes that represent the effects-annotations as explained in the previous section. The lattice defines the internal representation and relation between effects, such as sub-effecting. The domain defines how to transform annotations of effects into the internal representation used

by the lattice.

Every effect domain is represented by extending an abstract class called `EffectDomain`. Every domain has a lattice that represents the actual privileges and the relations between them. The effect domain classes are mostly identical in both plugins save for the implementation of the *eff* function used by the effect inference plugin, and the $\widetilde{\textbf{adjust}}$ and $\widetilde{\textbf{check}}$ functions used by the effect checking plugin.

Every effect domain must declare how to translate the set of scala-annotations into the internal representation of an effect and vice-versa. To do that the effect domain must declare the function `parseAnnotationInfo` to filter from the function annotations the effect-annotations of the domain. The function `toAnnotation` must be implemented to do the inverse translation, from the internal representation of effects to annotations.

Effect inference

In the domain of the effect inference plugin the function `computeEffectImpl` can be overridden to define the behavior of the function *eff* of the previous chapter. The function `computeEffectImpl` receives a node of an AST and an EffectContext, and it returns an Effect for that node. Consider the reference domain and the function *eff* defined as

$$\mathit{eff}_{alloc} = \{\texttt{@alloc}\}$$

$$\mathit{eff}_{write} = \{\texttt{@write}\}$$

$$\mathit{eff}_{read} = \{\texttt{@read}\}$$

and the implementation of the function *eff* over the Reference Domain is:

```
override def computeEffectImpl(tree: Tree, ctx: EffectContext): Effect =
    tree match {
        case Apply(fun, args) if isAlloc(fun.symbol) =>
          RefEffect(allocClass.tpe)
        case Apply(fun, args) if isWrite(fun.symbol) =>
          RefEffect(writeClass.tpe)
        case Apply(fun, args) if isRead(fun.symbol) =>
          RefEffect(readClass.tpe)
        case _ =>
          super.computeEffectImpl(tree, ctx)
    }
```

The function `computeEffectImpl` matches the node `tree` with an application where the function being applied is either alloc, write or read. If there is a match, then the function returns the associated effect with each case. If there is no match, then it calls the default function defined on EffectDomain.

Effect checking

In the effect checking plugin, the domain allows the user to override the functions $\widetilde{\textbf{adjust}}$ and $\widetilde{\textbf{check}}$ by implementing the functions `adjust` and `neededEffect` respectively.

The function `adjust` is in charge of adjusting the set of effect privilege on the nodes of the AST. The function receives as arguments the AST node on which the privilege set of its children will be modified. The function `adjust` defined in the `EffectDomain` base class matches the AST node as follows:

- The AST node is a typed node and has effect annotations, i.e., is an effect ascription. Then its children are adjusted with the effect annotations found.
- The AST node is an anonymous function. Scala does not store the type of an anonymous function on its symbol and consequently it does not have effect annotations. Then for anonymous function the effect is inferred recursively and then the children are adjusted accordingly. Special cases must be considered given that the effect system inserts runtime checks and casts of effect as lambdas or anonymous functions which may not type check.
- The AST node is a function definition, a class definition or a module definition. The privilege set of the children of the node are adjusted with the effects extracted from the annotations of the definition. A special case is considered to adjust the privilege set of the argument with default values.
- The AST node is an application. No special cases are done for an application except for some overriding of effect privileges on the children of the node in case of a match with an external effect specification.
- The AST node is none of the above. The privilege set of the children of the node are adjusted with the same privilege set of the current node. This way we are propagating the same effects on the sub expressions, i.e., the function $\widetilde{\textbf{adjust}}$ is not defined for that type of expression.

This function can be overridden in each effect domain implementation

The function `check` is in charge of testing if the AST node has the right set of effect privilege given its context. The function obtains the current privilege set Ξ and the necessary set of effect privilege to fulfill its requirements Ξ'. Then the function checks consistent subcontainment $\Xi' \sqsubseteq_{\sim} \Xi$ giving an error if the property does not hold.

A special case is considered for applications. If the AST node is an application then an extra check is done to test if the effect annotated on the type of the application is consistent subcontained on the privilege set Ξ. Before returning errors, the function applied is tested against the set of polymorphic effects given the context. If the name of the applied function is a parameter of an enclosing effect-polymorphic function, then no extra errors are propagated. The insertion of cast of effects envelops applications into lambdas, therefore pushing the application AST node down in the tree, tagging the node to not be re-check again.

A simplified version of the function `check` on the EffectDomain can be implemented like this:

```
def check(tree: Tree): EffectErrors = {
    val xi = getEffect(tree) //get the current privileges
    val needed = neededEffect(tree) //get the necessary privileges
    needed.map{ eff =>
       if(xi < eff) // check consisten subcontainment
           .... //return errors
     }.getOrElse{checkApplication(tree)}
}
```

The function can be overridden in each effect domain implementation. For instance, for the reference domain, the function that calculates the needed privilege set can be implemented as:

```
override def neededEffect(tree: Tree): Option[lattice.Effect] =
    tree match {
      case Apply(fun, args) if isAlloc(fun.symbol) => Some(RefEffect(allocClass.tpe))
      case Apply(fun, args) if isWrite(fun.symbol) => Some(RefEffect(writeClass.tpe))
      case Apply(fun, args) if isRead(fun.symbol) => Some(RefEffect(readClass.tpe))
      case _ => None
    }
```

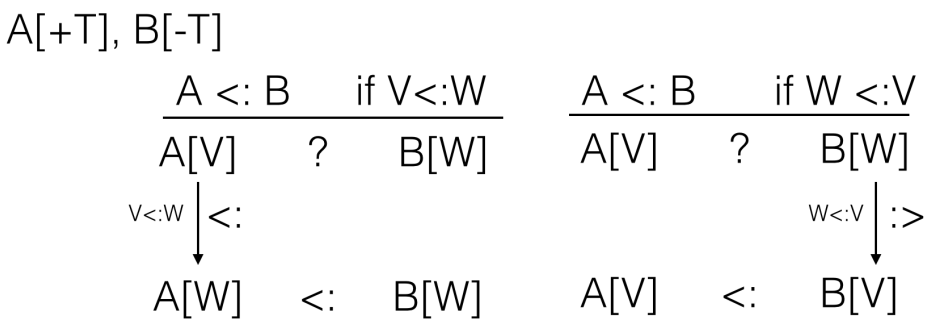
Thanks to pattern matching, the name of the function can be obtained and tested to decide which set of effect must be checked.

5.2.3 The Lattice

The lattice implements the internal representation of effects for a given domain. The lattice contains all the classes representing the privilege sets, the relations between the effects, and how to transform the effects into a runtime representation.

It is mandatory to define the top, bottom and unknown effect of the lattice. Also, the definition of binary operations such as join, meet, less than equal between the effect internal representation are needed.

Let us recall that when working with effect and type parameters, the variance of the type parameter can be covariant, contravariant, invariant and *bivariant*. Bivariant is defined to represent the subtyping relation between effects with type parameters which are covariant and contravariant. For instance, given the effects $A[+T]$ and $B[-T]$ and the relation $A[T] <: B[T]$, to say that an element $A[V] \in A[+T]$ is subtype of $B[W] \in B[-T]$, it is necessary to relate the type parameter from A with the one of B. That relation holds if some subtyping relation between V and W exist as the image shows:



Consistent Subcontainment The subtyping relations between sets of effect privileges are represented in each privilege class by overriding the $\leq$: operator. The relation *some* $\leq$: *other* is implemented as a method of the effect privilege classes by a pattern matching analysis over the "*other*" class. In case there are type parameters, they are analyzed given the covariant, contravariant, invariant or bivariant rules.

5.2.4 Effect Annotations

Static representation

The representation of the effects as annotations must be implemented for each domain. An annotation representing an effect can be any class, including classes with type parameters, depending on each implementation. Each class implementing the annotation of an effect must extend the class *Effect*.

For instance, the representation of the lattice `@C` <: `@B[+T]` <: `@A` is:

```
case class C() extends Effect
case class B[+T]() extends Effect
case class A() extends Effect
```

Case classes are used to easily compare effects annotation by comparing the case classes because comparing two instantiations of a class which are not case classes is always false by default.

Dynamic representation

To implement gradual effects a runtime representation of the effects must be implemented. This representation is used in runtime to dynamically check effects. The static representation is used on effects without type parameters. Effects with type parameters must define a special runtime representation. Every runtime implementation of an effect must extend the class *EffectsWithGeneric*.

The class *EffectsWithGeneric* enforces each implementation to define how subtyping is defined over the different effects of the domain. The definition of subtyping is necessary because of the different types of variance on the type parameters and the subtyping relations between effects. It is also necessary because the type parameters are lost at runtime thanks to type erasure. The runtime representation of a type parameter is preserved by using the name of the class as a String. At runtime the name of the class is converted to a type using reflection.

For example, the implementation of the effect `@B[+T]` can be:

```
case class runtimeB(tps: List[List[String]]) extends EffectsWithGeneric{
  override val privilegeName = "B"
  override def lte(b: Effect, c: String => String): Boolean = {
    b match {
      case _: runtimeB => ...  // covariant check of type parameters
      case _: A => true
      case _ => false
    }
  }
}
```

The class receives a list of list of strings as argument which represent the type parameters of the effect. For instance, consider the effect `@io[+T,-V]` which receives two type parameter. When joining two effects `@io[A, C]` and `@io[B, C]` the effect `@io[A | B, C]` is obtained. The type parameters of the joined effect is represented in the runtime representation as `List(List("A", "B"), "C")` . To define the representation shown at runtime errors of the effect the value `privilegeName` can be overridden with the name of the effect.

5.2.5 Working with multiple domains

The class `BiEffectDomain` permits the programmer to work with more than one effect domain. The class `BiEffectDomain` extends the class `EffectDomain`, taking two domains as argument, and reimplements all the methods of `EffectDomain` by recursively calling the same method of the domains of its argument, which can also be BiEffectDomains.

5.2.6 Effect checking at runtime

Having gradual effects on the system means that some runtime checks of effects must be done. The AST nodes are modified to insert the casts of effects, runtime checks of effects and runtime restriction of effects.

The checks at runtime are made using *DynamicVariables* which provides a binding mechanism where the current privilege set is found through dynamic scope, but where access to the variable is resolved through static scope. The dynamic variable used to check effects and

the functions associated are found in the object *RuntimePrivileges*.

The object RuntimePrivileges adjusts and checks the set the of effect privileges throughout the code using the runtime representation of each effect privilege. The blame of effects is also implemented in this object.

When the check of effect privileges produces a runtime error, the system tracks the innermost restriction of privilege set that made the check function fail. The innermost restriction which made the check failed is called the *localized restriction*. The system keeps propagating the error upwards, searching for a restriction where its context fulfil the original requirements. The first restriction to make the `has` command failed is called the *outermost restriction*. Finally, a runtime error is thrown indicating the cause of the error about the localized and the outermost restriction that made fail the check. If no outermost restriction is found, for instance if the initial privilege set made fail the check, then the cause of the error is the only localized restriction.

5.2.7 Modifying the code

Even though the type rules indicate that the modification to the code is done at runtime (see evaluation rules), the actual modification is done statically during compilation. Altering the compiled source code during runtime would imply modifying the Java Virtual Machine. During compilation, eta-abstraction is applied on variables to encapsulate functions being applied inside lambdas, allowing the insertion of `has` and `restrict` commands inside the newly created lambdas. For instance, let us consider a high-order function f applied with a function g where some runtime code must be inserted. The transformation can be for example:

$$f(g) \Rightarrow (\lambda x_0 \,.\, \texttt{has\{write, read\}}\ f((\lambda x_1 \,.\, \texttt{restrict\{read\}}\ x_0(x_1))))(g)$$

The modification of an AST node is done after the typer phase by the `insertHas` function.

Lazy insertion of code Given that the effect checking plugin runs after the typer phase, every new node inserted on the AST must be typed. The typing of the new AST nodes can be done manually by specifying every type and effect, or can be done automatically by using commands provided by the Scala compiler. Those commands that automatically type an AST node need some extra effort when the new node has children that needs to be typed as well. Lazy insertions of the commands `has` and `restrict` are done to automatically type the new nodes without the need to manually type every new inserted AST node.

Inserting Has

To check the need of inserting the `has` command, a different privilege set is calculated. The difference of effect is calculated between the strict privilege set required by the AST node

according to the type rules, and the strict privilege set found on the context. If the "delta" of privileges is not empty, the `has` command is inserted by altering the AST node.

For instance, when compiling the function

```
def unknownContext: Unit @unknown = {
  { ... } : @read
}
```

The transformed code looks like this:

```
def unknownContext: Unit @unknown = {
  RuntimePrivileges.dynCheck(List(read()), ...){
   ...
  } : @read
}
```

There is an ascription to `@read` in a context where the allowed privilege set is unknown. At runtime a check must be done to test if `@read` is present in the dynamic privilege set.

Inserting Restrict

There are two instances where the `restrict` command is inserted. The first one is on ascription of effects, and the other when inserting cast of effects. In both cases, the `restrict` command is encapsulated on a previous `has` command. The insertion of `restrict` is lazy to take advantage of the typer of the Scala compiler.

For instance, consider the function `unknownFun` typed $\texttt{Int} \xrightarrow{\{\iota\}} \texttt{Int}$ and the following code

```
unknownFun(1) : @read
```

The transformed code looks like this:

```
RuntimePrivileges.restrict(List(read()), ...){
    unknownFun(1)
} : @read
```

Where `restrict` modifies the dynamic variable that represents the privilege sets at runtime.

Inserting Casts

If the AST node is an application then it then tries to insert a cast of effects by calling the function `buildCastedTree` .

The function `buildCastedTree` receives an AST node as an argument. The application, let us say $e_1(e_2)$, is decomposed on e_1 and e_2. The latent effects Ξ_1 of e_1 are extended with its relative effects. For example, if e_1 is being effect polymorphic on its argument `someFunction`, i.e., the definition of e_1 has the effect annotation `@pure(someFunction)`, then the compiler needs to add to the latent effects the effects that `someFunction` might produce according to the argument e_2. A cast of effects is inserted if Ξ_1 or the current privilege set found in the context Ξ contain the unknown effect.

When a cast insertion is needed, the application AST node is then transformed using the function `insertCast` passing as argument e_1, the extended latent effect of e_1, the expression e_2 and the privilege set on the current context. The function `insertCast` is a recursive function that receivesthe expression being applied as argumment, the expression being applied, and the privilege set Ξ_1 and Ξ_2 being casted. Given that functions can receive more than one argument, the expression e_2 is a list of expressions. On each of the arguments of e_1, a cast is inserted recursively over the arguments by calculating the privilege set from the type of e_1 and e_2. The function then does a lazy insertion of `has` and `restrict` commands over the expression e_1. The cast is made by encapsulating the invocation inside anonymous functions.

For example

```
def unknownThatWrites(x: Int): Unit @unknown = {
      write(1)
}
```

Given that the privilege set allowed for the application `write(1)` is ¿, and the function `write` produces `@write`, then a cast is needed to be inserted to check if the privilege `@write` is present at runtime. The transformed code of the cast insertion looks like this:

```
((x0: Int) =>
    RuntimePrivileges.dynCheck(List(write()), ...){
         write(x0)
    }
})(1)
```

For a more complicated example, let us consider the functions

$$\texttt{f} : (h : \texttt{Int} \xrightarrow{\{@read, @alloc\}} \texttt{Int}) \xrightarrow[\{\texttt{h}\}]{\{@pure\}} \texttt{Int})$$

$$\texttt{g} : \texttt{Int} \xrightarrow{\{@read, ¿\}} \texttt{Int}$$

The function `f` is being effect polymorphic on its argument `h`. Consider the following application with the set of initial privileges {¿}

```
f(g _ )
```

The cast $\langle\langle(\texttt{Int} \xrightarrow{\{@read, ¿\}} \texttt{Int}) \xrightarrow{\{¿\}} \texttt{Int} \Leftarrow (h : \texttt{Int} \xrightarrow{\{@read, @alloc\}} \texttt{Int}) \xrightarrow[\{\texttt{h}\}]{\{@pure\}} \texttt{Int})\rangle\rangle$ must be inserted. The simplified version of the transformed code is

```
(x00 => {
    RuntimePrivileges.dynCheck(read(), ...)
    f((x10 => RuntimePrivileges.restrict(List(read(), alloc()), ...) {x00(x10)} )
})(g _ )
```

5.2.8 Customizable Effects

Using the DSL, the Domain and the Lattice are built automatically from a file with extension `.prg`. It allows the user to define an effect discipline by specifying the effect lattice and the external effect specifications where the effects must be set, added or removed. The file must be divided in three sections: the effect privilege section, the external effect specifications sections and the lattice section as defined in section 4.3.

Uncheck The effect system will infer "top" as effect for all functions defined in external libraries which are not be compiled using the effects plugin. When tagging a function as "unchecked" it will not produce effects, i.e., it will be typed as `@pure`. For example, if it is known that the collection library of Scala does not produce effects in a particular discipline, then it can be annotated globally as "unchecked" to avoid overestimation of effects.

To tag a function as "unchecked" it must be declared as a compiler option. For example, in a *sbt* file of a project

```
scalacOptions += "-P:effects:unchecks:org.joda.time.*"
```

means that all the function and classes inside the package "org.joda.time" will produce no effects.

Code generated The DSL generates the effect discipline with the information specified in the "privileges" and "external effect specifications" section. The most important function to be generated is the function **update** defined in the previous chapter.

The function `updateEffect` This function receives a node of the AST and a privilege set as arguments and returns a new privilege set. The privilege set returned is used to override the inferred effect of a function definition. This function is useful to set the effect for function definitions that have type annotations but no effect annotations. Current limitations of the effect inference plugin only allow the compiler to infer the effects of a function definition only when its type is also being inferred. Function definitions that have type annotation but no effect annotation are annotated with the effect "Top" the biggest possible for every domain. The overestimation of the effect over an unknown situation can be addresses by updating the function `updateEffect` for certain functions.

Suppose we have a function `insertDB` that insert some `Int` into a database. Unfortunately this function have type annotation but no effect annotation. We do not want the type

checker to infer that this function has the latent privilege set {`@write`, `@read`, `@alloc` }.

```
def insertDB(x: Int): Unit = {
  DB.insert(...) //produces the effect @write
}
```

One option is to annotate the effect manually which can be tedious if the same problem repeats multiple time. The other option is to override the `updateEffect` function to set the annotation of the function to be `@write`:

```
override def updateEffect(tree: Tree, eff: lattice.Effect) = {
    def obtain(tree: Tree): Option[lattice.Effect] = {
      val fun = tree
      val targs = tree.symbol.typeParams
      val DefDef(mods, name, tparams, vparamss, tpt, rhs) = tree
        if("""insertDB""".r.findAllIn(buildPath(fun.symbol)).hasNext){
          Some(RefEffect(Write()))
        } else eff
      }
      tree match{
        case _: DefDef => obtain(tree)
        case _ => eff
      }
  }
```

By matching the node of the tree with a function definition `DefDef`, then matching the name of the function to be insertDB, an effect `RefEffect(Write())` is returned that represents lattice representation of the effect annotation `@write`. The match is done using regular expressions, allowing generalizing the solution for multiple functions.

5.3 Limitations

5.3.1 Working with External Libraries (unanotated)

External libraries without effect annotations are inferred to produce the biggest privilege set or "top" by default. This may overestimate the privilege set of an application. Functions on external libraries will be called unknown functions.

Fully annotating and compiling the external library using the gradual effect system

This is the most natural solution but also the most difficult to execute. Sometimes the source code is not available and consequently this solution is unviable. Even with the source

code, compiling an external library may produce the same problem about external libraries recursively.

This solution may be preferred when using an external library that is small and the code is available.

Setting the latent effect of unknown functions as ¿

One can decide to tell the compiler to infer the effect of an unknown function as ¿. The problem with this solution is that to preserve soundness, runtime checks may need to be inserted in the body of the unknown functions. This implies that the external libraries must be recompiled using the plugin to insert the runtime checks which may result in the same problem recursively.

Override the latent effects using the DSL

In the DSL the user can specify the latent effect of a unknown function. This is useful when the user knows that the unknown function does not produce effects on the working disciplines. This solution is the one that gives the user most control and that is also the easiest to implement.

5.3.2 Other limitations

The gradual effect system currently has two problems: working with *Partial Functions* and an error using *Companion Objects*.

When using Partial Functions the effect inference plugin will produce a compiler error that is not yet solved as it depends on the plugin developed by Rytz *et al*.

Redefining the companion object of a `case class` and referring to the object defined on the `case class` will produce a compiler error in the effect inference plugin. For example,

```
case class Foo(id: Int)
object Foo{
      val x = Foo.apply _
}
```

will produce a compiler error that Foo is already defined. Of course, not using the plugin gives no errors.

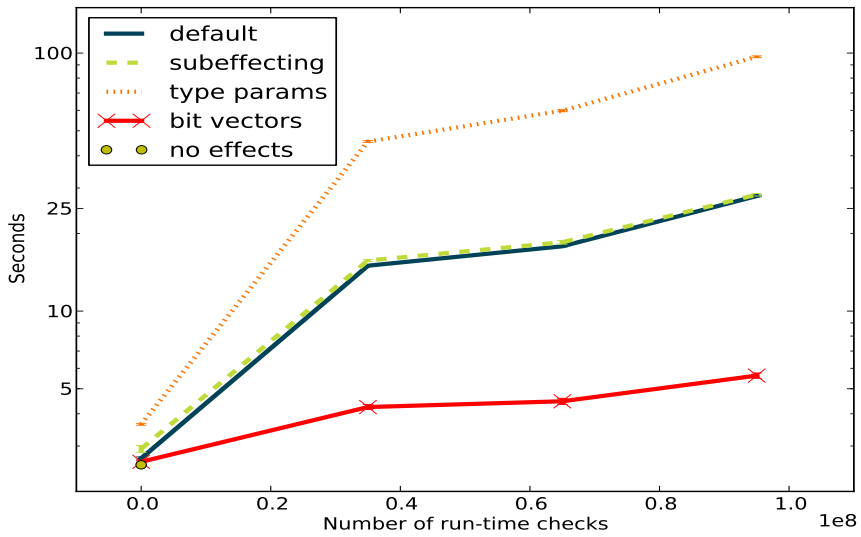


Figure 5.3: Impact of dynamic checks on execution time (logarithmic scale).

5.4 Benchmark

We now report on the cost of dynamic effect checks introduced by the gradual effect system in Scala. The experiments are run on an Apple computer with Mac OSX 10.9.5, 2.3 Ghz Intel Core i7 processor, 16GB of RAM, and flash storage. We run Scala version 2.10.4.

We consider a small program that includes effect ascriptions, polymorphic functions, function applications, and a custom-made `map` function to which we pass as arguments a list of ten integers and different kinds of functions: pure anonymous functions, effect ascripted functions and effectful functions.

We consider 4 versions of the same program, differing only in the number of runtime effect checks they induce. The first version of the program is fully annotated, hence its execution does not produce any runtime effect check. The second version has mostly unknown annotations, resulting in 95 dynamic checks (`has`) as well as 67 context adjustments (`restrict`) per run of the program. Finally, we use two intermediate partially-annotated versions, which produce 35 and 65 dynamic `has` checks per run, respectively (both produce 36 `restricts` per run).

A *benchmark run* consists in executing each of these programs one million times. For each program, we perform 20 benchmark runs, and report on the variance and average of all runs. Also, to avoid the noise of starting up the virtual machine, each experiment warms up the JVM by running the program 100.000 times. The results of the benchmark are presented in Figure 5.3, plotting the total number of runtime checks in each benchmark run with the corresponding time in seconds.

In addition to this baseline experiment (denoted "default" in Figure 5.3), we perform several other complementary experiments. First of all, we tested the piece of code without

the effect system, in order to measure the raw overhead of using the plugin. Also, in order to assess the impact of the complexity of the effect discipline on performance, we measure a scenario with subeffecting, and another with additionally type parameters. Lastly, we perform an experiment in which we use bit vectors instead of lists of objects for the runtime representation of privilege sets, so dynamic effect checks boil down to bitwise operations. More precisely, each bit of the byte represents a specific effect privilege, where 1 means the privilege is present, 0 otherwise.

Results. Table 5.1 presents the detailed measurements of the benchmarks.

	# `has`	0			35M			65M			95M		
	# `restrict`	2M			36M			36M			67M		
scenario		avg	std	ovhd	avg	std	ovhd	avg	std	ovhd	avg	std	ovhd
default		2.686	0.036	1.06x	14.994	0.104	5.91x	17.828	0.109	7.03x	27.926	0.202	11.0x
subeffecting		2.925	0.091	1.15x	15.703	0.152	6.19x	18.485	0.306	7.29x	28.265	0.321	11.4x
type params		3.641	0.031	1.44x	45.403	0.407	17.9x	59.783	0.537	23.6x	96.753	0.786	38.2x
bit vectors		2.605	0.022	1.03x	4.247	0.095	1.68x	4.472	0.122	1.76x	5.623	0.135	2.29x
no effects		2.535	0.025	1.00x	-	-	-	-	-	-	-	-	-

Table 5.1: Detailed benchmark results

The runtime overhead of using the effect system, without runtime effect checks, is only 5%. The overhead comes from the `restrict` operations, which track the dynamic current set of privileges during execution. When everything is fully annotated, `restrict` s are still introduced to track the initial privilege set, and every time an effect ascription is encountered. Note that it would be possible to optimize this scenario further through a flow analysis that determines whether the current privilege set is eventually used or not, thereby avoiding some (if not all) `restrict` operations.

The cost of dynamic effect checking in the "default" scenario increases significantly, starting with 15% overhead and going up to 10x slower. Remember that this is using the default representation of privilege sets as lists of privilege objects. Adopting the bytes representation of privilege sets is a drastic improvement, with an overhead starting at 2.7% and reaching only 121% in the worst case.

Adding subeffecting, while provoking more involved checks, does not induce any noticeable overhead compared to the default scenario. It is certainly possible to design an optimization for subeffecting relying on a byte-level representation, although this is left as future work.

Using type parameters in the effect lattice badly affects performance. Recall that, because type parameters are erased when Scala is compiled to Java, the plugin uses strings and reflection to compare type parameters at runtime. In fact, the raw cost is much worse than the one presented here, because we have implemented a caching mechanism to reduce the overhead of reflection. Even with this optimization, the execution is 38x time slower.

To summarize, we find the results particularly encouraging for a first practical implementation of a gradual effect system. The observed performance of the implementation respects the "pay-as-you-go" motto of gradual typing: the overhead for a fully static program is acceptable, and there is a progressive reduction in the overhead as the "static-ness" of the

effect discipline augments. In addition, there are pending optimization opportunities to be explored in each of the scenarios. This being said, the high impact of type parameters in the lattice is probably unavoidable considering the limitations of the JVM in this respect.

5.5 Summary

In this chapter we have shown how we implemented the customizable effect system in the Scala programming language as two compiler plugins. In section 5.1, we explained that a type rule of the gradual effect system described in TGE is non-deterministic. We have shown how a bidirectional effect system solves this problem by using effect inference to annotate function definitions to later apply a fully deterministic effect checking system.

In section 5.2 we explained how we implemented the solution as two compiler plugins: the effect inference plugin and the effect checking plugin. The effect inference plugin is an extension of the work done in PES with support for gradual effects and the DSL. We focus mostly on the effect checking plugin.

In section 5.2.1 we have shown how we introduce effects annotations into Scala by using scala-annotations in types. Then, in section 5.2.2, we have shown how disciplines are built, the functions that must be defined and some examples of the most important functions.

In section 5.2.3 we briefly explained how the lattice is implemented and what operations must be defined. We have shown how type parameters can be related when used on effect lattices.

In section 5.2.4, we explained the static and dynamic representation of effects as scala-annotations. We have shown that when using type parameters on effects, the dynamic representation must consider special cases given type erasure of type parameters.

In section 5.2.5 we have shown how to work with multiple disciplines at the same time by using the function `BiEffectDomain`. Section 5.2.6 explained how effects are adjusted and checked at runtime. We also explained that a "blame" system is implemented to indicate to the programmer where the problem was produced.

In section 5.2.7 we have shown that most of the modification of the source language is lazy to take advantage of the Scala typer. We have given some examples of inserting `has` and `restrict` commands.

In section 5.2.8, we focused on explaining how the customizable effect system interacts with the framework by the automatic generation of code.

In section 5.3, we explained some limitations of the solution when working with external compiled libraries and how they can be addressed with the use of the DSL.

We finalized this chapter with section 5.4, presenting the benchmarks of the implementation. We show that the cost of inserting dynamic cast is greatly increase when working with

effects with type parameters. In the following chapter we will show usage examples of the solution.

Chapter 6

Examples

In the previous chapter We have described the implementation of the customizable effect system in Scala as compiler plugins. In this chapter we will show some basic examples of the framework, and then we will show two real cases of the DSL.

6.1 The Combined Effect System

For the following examples let us consider the reference domain and the effects `@write`, `@alloc` and `@read`. Suppose the function `write` is typed $\texttt{Int} \xrightarrow{\{\texttt{@write}\}} \texttt{Unit}$, `alloc` is typed $\texttt{Int} \xrightarrow{\{\texttt{@alloc}\}} \texttt{Unit}$, and `read` is typed $\texttt{Int} \xrightarrow{\{\texttt{@read}\}} \texttt{Unit}$.

Introducing effect annotations can be done like this:

```
def bar1(x: Int): Int  @unknown = {
  write(1)
  x
}
def foo1(x: Int): Unit @alloc @write = {
  bar1(2)
  alloc(1)
  write(2)
}
```

Where the function `bar1` has the latent set of effect privileges $\{¿\}$, and the function `foo1` has the latent privilege set $\{\texttt{@alloc}, \texttt{@write}\}$.

When a function has no type and effect annotations, then the type and effect is inferred:

```
def inferAllocAndRead(x: Int) = {
  if(condition)
    alloc(1)
```

```
      else
        read(2)
      x
    }
```

The function `inferAllocAndRead` will be inferred to have the return type `Int` and the latent privilege set {`@alloc`, `@read`}. When a function is annotated with a privilege set, the effect of the body is inferred and compared with the annotation.

```
    def smallerAnnotation(x: Int): Unit @alloc = {
      inferAllocAndRead(2) // type error
    }
```

The compiler gives a static compiler error on `inferAllocAndRead(2)` because `inferAllocAndRead` may produce the effects {`@alloc`, `@read`} which are bigger than the one annotated {`@alloc`}.

To delay the error on `inferAllocAndRead(2)` to runtime, an ascription can be made like this:

```
    def smallerAnnotation(x: Int): Unit @alloc = {
      inferAllocAndRead(2) : @unknown
    }
```

Even though `inferAllocAndRead` may produce the effects {`@alloc`, `@read`}, they are consistent with {¿}. And as {¿} is consistent with {`@alloc`} this expression is statically validated by the effect system. If the function `smallerAnnotation` is applied, then a runtime error is produced by the cast inserted by the type system in the invocation of `inferAllocAndRead` function.

6.1.1 high-order functions

Let us consider now a high-order function that receives as argument another function typed $\texttt{Int} \overset{\{\texttt{@read},\texttt{@alloc}\}}{\longrightarrow} \texttt{Int}$

```
type ReadAndAlloc = (Int => Int) { def apply(x: Int): Int @read @alloc }
def highOrder(f: ReadAndAlloc): Int @read @alloc = {
      f(1)
}
```

Invoking `highOrder` with a function that may produce the effect `@write` would lead to a static error.

```
def h(x: Int) = {
      if(condition) write(1) : @unknown else 2
}
highOrder(h _) //should type check, but fail at runtime
```

But once again, when invoking `highOrder` with a function typed as `Int` $\xrightarrow{\{¿\}}$ `Int`, it would produce a runtime error on the expression `f(1)`.

Let us consider now a function where the body produces `@alloc` but is annotated as {`@unknown`}, and an initial set of privileges {`@read`, `@alloc` }

```
def unknownThatAlloc(x: Int): Unit @unknown = {
      alloc(x)
}
unknownThatAlloc(2): @read @alloc @write //should fail
unknownThatAlloc(2) : @read //should throw a runtime
```

A static error is produced when an application of `unknownThatAlloc` is ascripted to the privilege set {`@read`, `@alloc`, `@write` }, because {`@read`, `@alloc`, `@write` } is not consistent subcontained in the initial privilege set {`@read`, `@alloc` }. When the application is ascripted to a valid set {`@read` } the execution throws a runtime error because the effect `@alloc` is being produced but the set of authorized privileges for that context is only `@read`.

Now let us consider a trickier example given the same initial privilege set {`@read`, `@alloc` }:

```
def functionThatReads(x: Int): Int @read = x
def pureContext(x: Int) : Unit @pure = {
    functionThatReads(1) : @unknown
}
pureContext(1) //should fail at runtime??
pureContext(1) : @pure //should fail at runtime
```

The function `pureContext` is annotated with the latent effects {`@pure`} but the body produces {`@read` }. A static error would be produced, but the effectful operation is encapsulated on an ascription to the {¿} set. A question that arises is if the invocation of `pureContext` should fail at runtime. The answer is no, because the effect-annotations on the function signature do not alter the dynamic privilege sets, the effect-annotations on function signatures only statically check if the effects produced by the body conform with the effect-annotations. On the contrary, when the invocation of `pureContext` is encapsulated on an ascription to {`@pure`}, then it is correct to throw a runtime error.

6.1.2 Effect-polymorphism

Let us consider the initial privilege set {`@read`, `@alloc` } and a high-order function that is effect-polymorphic on its argument:

```
type producesWrite = Function[Int,Int]{def apply(v1: Int): Int @write @pure }
def polyOnGFunction(g: producesWrite): Int @pure(g) = {
    g(10)
}
```

The type of the argument that the function `polyOnGFunction` is being effect-polymorphic has the latent effect {`@write` }. Now let us consider two new functions:

```
def writeFunction(x: Int): Int @write @pure = {
  write(10)
}
def pureFunction(x: Int): Int @pure = {
  x
}
```

Even though the body of `polyOnGFunction` is invoking its argument, applying the function `polyOnGFunction` to a pure function gives no type errors:
`polyOnGFunction(pureFunction _)`
As `polyOnGFunction` is effect-polymorphic on the argument, the privileges of its argument are checked when `polyOnGFunction` is being applied. On the other hand the application
`pureOnGFunction(writeFunction _) : @unknown`
gave a runtime error because its argument produces {`@write` }.

When a function is being effect-polymorphic on an argument that has unknown latent effects

```
type unknownLatent = Function[Int,Int]{def apply(v1: Int): Int @unknown }
def polyOnUnk(g: unknownLatent): Int @pure(g) = {
  g(10)
}
```

invoking `polyOnUnk` with a function that produces an unauthorized effect:
`pureOnUnknownFunction(writeFunction _) : @unknown` produces a runtime error. Clearly, passing a function with unknown latent effects, introduces the intuitive checks at runtime:

```
def unknownFunctionThatWrites(x: Int): Int @unknown = {
  write(10)
}
polyOnUnk(unknownFunctionThatWrites _) : @unknown //throws a runtime error
```

6.1.3 Classes and Objects

Working with classes is straightforward. Let us consider an initial privilege set {`@read`, `@alloc` } and the following class definition:

```
class A{
  def writeFun(x: Int) = {
    write(x)
  }
}
```

```
val a = new A()
a.writeFun(9) //static error
a.writeFun(0) : @unknown //throws a runtime error
```

The constructor of class `A` does not produce side-effects. Invoking effectful operations that are members of class `A` behave equally to function invocation. Classes and objects have the same behaviors.

```
object B{
  def writeFun(x: Int)= {
    write(x)
  }
}
B.writeFun(9) //static error
B.writeFun(0) : @unknown //throws a runtime error
```

A function can be effect-polymorphic on a member of a class like this:

```
def writeFunObj(a: A):Unit @pure(a.writeFun(\%)) = {
  a.writeFun(10)
}
```

Hierarchy of classes is supported:

```
class BaseClass{
  def fun(x: Float): Unit @read @alloc = {

  }
}
class C extends BaseClass{
  override def fun(x: Float): Unit @unknown = {
    write(x)
  }
}

class D extends BaseClass{
  override def fun(x: Float):Unit @read = {
    read(x)
  }
}
```

And as excepted, when applying `fun` to a instance of the class `C`, show throws a runtime error:

```
val c = new C()
//c.fun(10) //should throws a runtime
```

6.2 The Customizable Effect System

6.2.1 Customized IO

Let us consider the IO effect discipline with two effects: `@input` and `@output`. Printing into screen using the standard Scala library produces the effect `@output` and waiting for user input produces the effect `@input`. The DSL file is presented below:

```
name: IO
privileges:
    @input
    @output
    @noIo
lattice:
    top: @input @output
    bottom: @noIo
pointcuts:
    app scala.Predef.read* => set @input
    app println => set @output
```

The effect `@noIo` is defined to represent the absence of effects. In Scala, input from a user can be captured using the functions `readInt`, `readByte`, etc. The join point `scala.Predef.read*` is used to track input effects on functions from package "scala.Predef" that starts with the name "read".

Suppose that the programmer wants to track only when Strings are printed. The last external effect specification may be modified like this:

```
    ...
    app println(T <: String) => set @output
```

The user may want to tailor the effect discipline to specify the corresponding type of the IO operations.

```
name: IO
privileges:
    @input[+T]
    @output[+T]
    @noIo
lattice:
    top: @input[Any] @output[Any]
    bottom: @noIo
pointcuts:
    app scala.Predef.read*: T => set @input[T]
```

```
    app println(T <: String) => set @output[T]
```

A type parameter is added to each effect privilege. Top and bottom must be instantiated with the type parameter `Any` because `Any` is the root the Scala class hierarchy, i.e., Every class inherits directly or indirectly from this class. The external effect specifications associate the types of the arguments of the function with the type parameter of their corresponding effect.

6.2.2 Controlling database access on the Play Framework

Play is a web framework for Scala and Java. A Play application follows the MVC architectural pattern applied to the Web architecture. The MVC pattern splits the application into three layers: the model, the view and the controller. The model declares the representation of the information on which the application operates. The view renders the model into a user interface. The controller responds to events and may invoke changes on the model.

Usually the model uses a persistence storage mechanism to store the data, for example a database. The access to the database is generally encapsulated by the model, but nothing prevents accessing the storage directly from the view or the controller. We present a basic effect system defined with the DSL to control database access.

Let us suppose that to access the database the "Slick" library is used. Also, every model lies within the `models` package, every controller lies within the `controllers` package, and every view lies within the `views.html` package. This simple DSL will prevent the view to invoke a function that may produce a query in the database:

```
name: DatabaseAccess
privileges:
        @access
        @noAccess
lattice:
    top: @access
    bottom: @noAccess
pointcuts:
        def views.html.*.apply() => set @noAccess
        app scala.slick.backend.*.* => set @access
```

The discipline `DatabaseAccess` defines two effects: `@access` and `@noAccess` to represent database access. The first external effect specification is declaring that every view can not access the database, we want only the controller to access the database through the model. The second external effect specification is declaring that every function inside the package `scala.slick.backend` will produce the effect `@access`.

We could extend this DSL to track insertions on the database.

```
name: DatabaseAccess
privileges:
         ....
        @insert[+T]
lattice:
    top: @access @insert[Any]
    bottom: @noAccess
pointcuts:
        def views.html.*.apply() => set @noAccess
        app scala.slick.backend.*.* => add @access
        app models.*.insert(T) => set @insert[T]
        app scala.slick.*.insert(T) => set @insert[T]
```

The effect `@insert[+T]` represents the insertion of an object of type T into the database. Two new external effect specifications are added to declare that whenever a function called "insert" from a package inside the model package or the slick package, an `@insert` effect is produced instantiated with the type of the argument of the "insert" function. For example, let us consider a CRUD (Create Read Update Delete) controller named "UserController" that only does operations on the "User" model. To prevent the controller from doing database insertions of models that are not the "User" class, the body of the action of the controller can be embedded in an effect ascription:

```
def index = Action { implicit request =>
    ({
        ....
    }) : @insert[User] @access
}
```

This way, the body of the action will only be allowed to produce the `@insert[User]` and `@access`. Let us suppose now that an object is created to access the database for every model. For instance, the object called "Users" is responsible of executing all the database queries that correspond to the "User" model. The object "Users" should only insert "User" models into the database. This can be done by adding an extra external effect specification to the DSL:

```
pointcuts:
         ...
        def models.Users.* => set @access @insert[User]
```

Chapter 7

Conclusions

7.1 Contributions

In this work we have designed and formalized a domain-specific language that allows programmers to easily create and modify effect disciplines. The DSL aids programmers in adapting an effect discipline by easily creating application-specific effect disciplines tailored to the programmer needs. We have shown how external effect specifications can be used to "passively" introduce an effect discipline into an existing source code with minimal modifications to the source code.

We presented a formalization of an effect system that extended the work on gradual effect system of Bañados *et al* with support for effect polymorphism by expanding relative effects before the casts are inserted. We have also extended the gradual effect system with support for sub-effecting, in order to express complex lattices that take into consideration "ordering" between effects.

The solution is the first implementation of a gradual effect system which helps programmers easily adapt an effect discipline into existing projects by slowly annotating effects as they are needed. We have shown how the final effect system can be fully deterministic by following a bidirectional approach combining two steps: effect inference and effect checking. This implementation also aids programmers in empirically evaluating the claim that gradual effect checking improves the use and declaration of effect disciplines by programmers. Finally we have shown how this effect system can easily be integrated with the popular Play web framework for the Scala language.

7.2 Future Work

- Extend expressiveness of the DSL. The DSL is inspired on the work implemented on AspectJ. It does not support all of the features found in AspectJ that could be added in the future. It is also desirable to extend the DSL to support effect-polymorphic

annotations.

- Update the Scala plugin to the latest version of Scala. The current version of the Scala plugins work on version 2.10.x because it was the stable version when the project started. Now, Scala is on version 2.11.x and the plugin has not been tested on that version.
- Extend the effect system to not only work with effect as privileges or authorizations, but also as "requirements". The current effect system does not allow programmers to enforce the production of an effect, for instance, to throw an error if no "print" is found inside a function. In other words, it would be useful to change the consistent set by a "range set" indicating the minimal and maximal privilege sets required and authorized for a given expression respectively.

Bibliography

[1] F. Bañados, R. Garcia, and É. Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 283–295, Gothenburg, Sweden, Sept. 2014. ACM Press.

[2] C. Chambers. The Cecil Language Specification and Rationale: Version 2.0. 1995.

[3] W. R. Cook. On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572, 2009.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, Jan. 1977. ACM Press.

[5] R. B. de Oliveira. The Boo programming language, 2005.

[6] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the ACM Conference on LISP and Functional Programming (LFP '86)*, pages 28–38, 1986.

[7] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 231–245, New York, NY, USA, 2005. ACM.

[8] D. Marino and T. 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.

[9] E. Meijer and P. Drayton. Static Typing Where Possible, Dynamic Typing When Needed. 2005.

[10] L. Rytz. *A Practical Effect System for Scala.* PhD thesis, École Polytechnique Fédérale de Lausanne, Sept. 2013.

[11] L. Rytz, M. Odersky, and P. Haller. Lightweight polymorphic effects. In J. Noble, editor, *Proceedings of the 26th European Conference on Object-oriented Programming*

(ECOOP 2012), volume 7313 of *Lecture Notes in Computer Science*, pages 258–282, Beijing, China, June 2012. Springer-Verlag.

[12] J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

[13] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in Lecture Notes in Computer Science, pages 2–27, Berlin, Germany, July 2007. Springer-Verlag.