Customizable Gradual Polymorphic Effects for Scala

Matías Toro *
PLEIAD Lab
Computer Science Department (DCC)
University of Chile
mtoro@dcc.uchile.cl

Éric Tanter †
PLEIAD Lab
Computer Science Department (DCC)
University of Chile
etanter@dcc.uchile.cl

Abstract

Despite their obvious advantages in terms of static reasoning, the adoption of effect systems is still rather limited in practice. Recent advances such as generic effect systems, lightweight effect polymorphism, and gradual effect checking, all represent promising steps towards making effect systems suitable for widespread use. However, no existing system combines these approaches: the theory of gradual polymorphic effects has not been developed, and there are no implementations of gradual effect checking. In addition, a limiting factor in the adoption of effect systems is their unsuitability for localized and customized effect disciplines. This paper addresses these issues by presenting the first implementation of gradual effect checking, for Scala, which supports both effect polymorphism and a domain-specific language called Effscript to declaratively define and customize effect disciplines. We report on the theory, implementation, and practical application of the system.

Categories and Subject Descriptors D.3.1 [*Software*]: Programming Languages—Formal Definitions and Theory; D.3.3 [*Software*]: Programming Languages—Language Constructs and Features

Keywords Type-and-effect systems; gradual typing; effect polymorphism; Effscript; Scala

1. Introduction

Type-and-effect systems allow static reasoning about the computational effects of programs. Effect systems were originally introduced to safely support mutable variables in functional languages [17], but more recently, effect systems have been developed for a variety of effect domains, e.g., I/O, exceptions, locking, atomicity, confinement, and purity [1, 2, 6, 18, 19, 31, 32].

Despite their potential, effect systems are not widely used by programmers. Several recent developments have enhanced the state-of-the-art of effect systems towards making them more practical. In particular, Marino and Millstein developed a *generic* effect system that makes it possible to see many effect disciplines as instances of a general-purpose system for granting and checking privileges [27]. The generic effect system underlies the design of the Scala effect checker plugin [30], which also makes progress on the practical side by supporting a lightweight form of *effect polymorphism* [32]. More recently, in order to smoothly and progressively allow programmers to adopt effect typing, as well as circumventing the expressiveness limitations of effect systems, Bañados *et al* developed a theory of *gradual* effect systems [5]. A gradual effect system supports a sound combination of static and dynamic effect checking. The gradual effect system is formulated in terms of the generic framework of Marino and Millstein.

Unfortunately, to date there is no implementation of a gradual effect system. In addition, the theory of gradual effect checking does not consider effect polymorphism, which is crucial for handling common higher-order abstractions with enough precision [26, 32]. Furthermore, even though a gradual polymorphic effect system would be of both theoretical and practical value, its adoption would still be limited by the fact that effect disciplines tend to be too rigid to fit the specific needs of developers. Developers should be able to easily define an application-specific effect discipline, for instance declaring that they care about tracking input/output (IO) effects in certain parts of their programs, and to possibly consider certain IO operations such as login as harmless.

This paper addresses all these issues as follows:

- We develop the theory of gradual polymorphic effects, highlighting the challenges in combining lightweight effect polymorphism [32] and gradual effect systems [5] (Section 4).

* Funded by CONICYT-PCHA/Magíster Nacional/2013-22131048 and Fondecyt project 1150017.

† Partially funded by Fondecyt project 1150017.

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2814270.2814315

- We introduce the notion of *customizable* effect systems through a domain-specific language, dubbed Effscript, for defining effect disciplines (Section 5).
- We implement a customizable, gradual, polymorphic effect system for Scala as a compiler plugin (Section 6)
- We report on the application of the effect system and Effscript to enforce architectural constraints in the popular Play [39] web framework, and report on the performance of gradual effect checking (Section 7).

Section 2 provides a more detailed presentation of the issues at stake, and Section 3 gives a brief informal overview of our proposal. Section 8 discusses related work and Section 9 concludes. The implementation of the system, as well as the code of the examples and benchmarks, is available online at: `http://pleiad.cl/effscript`. The complete formal semantics and soundness proof can be found in a companion technical report [37].

2. Background and Motivation

We start by briefly introducing recent advancements in making effect systems practical, namely effect polymorphism and gradual effect checking, before motivating the need for an effect system that integrates both mechanisms and goes further in giving programmers direct control over the effect system.

2.1 Background

An effect system tracks the side-effects of program expressions. Values, including first-class functions and objects, are by definition effect-free. However, a function or method may produce effects when it is applied—these are called the *latent effects* of the function. To illustrate the basics of effect systems, we use Scala and its effect typing plugin, developed by Rytz [30]. We focus on the input/output (IO) effect discipline, tracking whether expressions produce some IO effect or not. In the Scala plugin, these are represented as annotations, `@io` and `@noIo`. The `print` function produces `@io` when applied, while the `reverse` function on lists produces `@noIo` (we also say that it is *pure* in the IO domain).

For instance, the following function is declared to have `@io` latent effects, because it *may* apply `print`:

```
def foo(x: Int): Int @io  = {
  if(x == 0){ print("zero!"); 1 }
  else 0
}
```

The type of `foo` is denoted $\texttt{Int} \xrightarrow{\texttt{@io}} \texttt{Int}$, with the latent effect on top of the arrow (in the general case, it can be a set of annotations). Note that annotating `foo` with the `@noIo` latent effect would be rejected by the type checker, as the call to `print` violates such a specification.

Generic effect systems. Marino and Millstein [27] noticed that many effect disciplines can in fact be seen as instances of a generic framework for effect typing. They develop the theory of a generic effect system, which captures the essence of effect systems in the form of a *privilege checking* system. Expressions can be seen as requiring privileges (dually, producing effects) to execute properly, and the language must include ways to grant such privileges, either through annotations (as above) or through dedicated constructs (as in the `try/catch` construct, which grants the privilege to `throw`).

Effect polymorphism. The generic system of Marino and Millstein has been adopted by Rytz and Odersky [30, 32] to develop a practical effect system for Scala. The key contribution of the Scala effect system is its mechanism for *Lightweight Polymorphic Effects* (LPE) [32], which conveniently supports higher-order programming patterns.

Consider the `map` function, which applies its function argument to each element of its list argument. Because the effects of applying `map` depend on the effects of the argument function, a non-polymorphic effect system has to conservatively consider `map` to have the maximum effects possible. Consequently, even benign uses of `map` in a context that requires purity are rejected by the type system.

LPE addresses this issue by allowing a function to declare that it is (effect) polymorphic in some of its arguments. For instance, we can define the `map` function as follows:[1]

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

The argument `f` has no declared latent effects, which is equivalent to annotating it with the maximal latent effects, $\top$. Consequently, any function can be passed as argument to `map`. The key to be able to apply `map` without always inferring that it produces $\top$, lies in the `@pure(f)` annotation: this annotation states that `map` is pure, modulo the effects of its argument `f`. The type system therefore determines the effect of an application `map(l,g)` by combining the specific effects of `map` (here, `@pure`) with the latent effects of the actual argument `g`.

Unchecked annotations. The benefits of effect polymorphism in LPE are still subject to the limitations of static type checking: in some scenarios, the conservative approximations of the system get in the way. For instance, consider the application `map(l,print)`:[2] the polymorphic effect system determines that it has the `@io` effect, *regardless* of whether the list `l` is actually empty! In such a case, `map` does not apply `print`, and hence the application could safely be considered to be pure.

[1] For conciseness, we consider that `map` only operates on lists of `Int`.

[2] To be precise, this is written `map(l, print _)` in Scala, to obtain the closure value of `print`. We omit these `_` in this paper.

To deal with such cases, which are frequent in practice [32], it is common for effect systems to introduce a way to (unsafely) bypass the effect system. Both Koka [26] and the Scala effect plugin provide such a mechanism. In Scala this is achieved using the `@unchecked` annotation. For instance, suppose a `spawn` function that takes a by-name argument (*i.e.* a thunk) that must be pure, then:

```
spawn { e : @unchecked @pure }
```

is accepted by the effect system, without further checks. Of course this is unsound if `e` does perform some effects.

Gradual effects. A sound approach to circumvent the rigidity of static checking is to rely on *gradual* checking. For instance, gradual typing [34] allows fine-grained safe interactions between statically-typed and dynamically-checked expressions. Gradual typing introduces an unknown (aka. dynamic) type annotation, which is used as the default for unannotated code. Statically, the gradual type system rejects only programs that surely go wrong, and accepts programs that may go right, but safeguards this flexibility by introducing runtime checks (casts).

Recently, Bañados *et al* developed a theory of gradual effect checking [5] (hereafter TGE). An unknown effect is introduced, denoted '¿' in the theory, and `@unknown` in the Scala code of this paper. For instance:

```
spawn { e : @unknown }
```

This program is accepted statically, as with `@unchecked`. The difference is that, during compilation, an *effect cast* is inserted just before every effectful operation in `e`. At runtime, if the flow of execution reaches such a check, a runtime effect error is thrown if the surrounding context does not grant sufficient privileges for the effectful operation to be performed.

2.2 Towards a Practical Effect System

Both effect polymorphism and gradual effect checking are valuable assets for a practical effect system. However, to date, there is no implementation of a generic gradual effect system, and the interaction between gradual effects and effect polymorphism have not been studied. A major contribution of this work is to explain how to combine both approaches, and to implement this system for Scala.

However, combining gradual and polymorphic effect checking is not enough for an effect system to be practical. As we discuss below, an effect system should also be easily customizable to suit the specific needs of developers.

Gradual polymorphic effect checking. While both the theory of gradual effect checking and the theory of lightweight polymorphic effects already exist [5, 30, 32], defining a gradual polymorphic effect system is more challenging than just merging the semantics of LPE and TGE together. There are subtle interactions to consider, as will be illustrated in Section 3.1, because one expects to be able to take an unannotated `map` function and use it polymorphically.

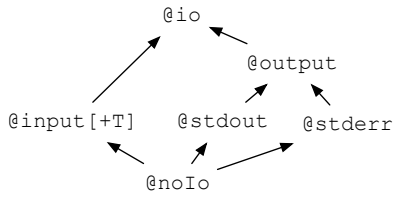

Figure 1. Refined lattice for the IO domain

Customizable effects. While effect polymorphism and gradual effect checking are important steps towards a practical effect system, the language-wide approach of effect domains can be too rigid in practice. Consider the IO effect discipline as implemented in Scala: *every* IO operation is considered effectful and must be taken care of by programmers. It is not possible for the programmer to track *only some* IO operations, for instance printing to the standard output stream, and to consider others, such as printing to the error output stream, as non-effectful. This issue was recently raised on the Scala users mailing list:[3]

> *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* IO operation as a side effect. Or, if I want to consider every IO 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.*

To which Martin Odersky responded:

> *Amr Sabry characterized side effects as "anything that makes order of execution observable" (my words). "Observable" is a term that's up to interpretation. In your case, since you decided that you do not care about log messages at all, it would be defensible to regard logging as side-effect free. How to tell that the type system is a different question.*

We aim at answering exactly this question, by providing a declarative means for users to *customize* effect disciplines to their specific needs. Furthermore, we are not aware of an existing implementation of a generic effect system in which defining a new discipline from scratch can be done easily and declaratively. So far, the only concrete implementation of a multi-domain effect system we know of is the Scala effect plugin of Rytz [30], which only supports a couple of disciplines and is not easily modified. This ties practitioners to the specific disciplines that were designed by the provider of the effect system.

Effect lattices. Designing a customized effect discipline can imply introducing a different taxonomy of effects from the one anticipated by the provider of the effect system. For

[3] https://groups.google.com/forum/#!topic/scala-user/3vG7VJlZIxg

instance, the Scala effects plugin treats the IO domain in a very simple binary fashion: `@io` vs. `@noIo`. In fact, as mentioned before, a developer may want the IO domain to classify `@stderr` and `@stdout` as separate effects. But the relation can be even more complex, yielding an arbitrary *effect lattice*, as depicted on Figure 1. In this example lattice, `@io` effects are first classified as `@input` or `@output`, which is itself further refined by `@stderr` or `@stdout`. Also, just like exceptions are effects parametrized by a type, the programmer may want to reason about which type of values are obtained through an input operation, expressed as the (covariant) `@input[+T]` effect.

An effect lattice expresses *subeffecting* [30], *i.e.* a partial order between effects: `@stdout` is a subeffect of `@io`, and `@input[String]` is a subeffect of `@input[Any]`. Subeffecting is supported in the effect system of Rytz *et al* (for exceptions), but not in the theory of gradual effects of Bañados *et al*. A customizable effect system should support the seamless specification of arbitrary effect lattices, suited to the specific needs at hand.

External effect annotations. The adoption of an effect discipline on an existing code base implies modifying the source code to introduce effect annotations, either on method signatures or as effect ascriptions. In the same way in which aspect-oriented programming [22, 24] provide means for localized specification of scattered behavior, it should be possible to specify effect annotations externally from the code base. This would better support customizable effects by making it easier to react to changes in the design of the effect discipline, such as refinements in the effect lattice.

3. Customizable Gradual Polymorphic Effects: A Brief Overview

This section gives an informal description of the proposed system, focusing first on the integration of gradual checking and effect polymorphism, and then presenting Effscript, a domain-specific language that allows programmers to declaratively define and customize effect disciplines.

3.1 Gradual Polymorphic Effect System

As alluded to in Section 2.2, naively combining effect polymorphism and gradual effects does not yield a practical system. We illustrate the challenges and solutions below.

Default latent effects of higher-order arguments. Consider again the `map` function, this time without effect annotations (equivalent to specifying it has an unknown effect):

```
1  def map(l: List[Int], f: Int => Int):
2    List[Int] @unknown = {
3    l match {
4      case Nil => Nil
5      case x :: xs => f(x) :: map(xs, f)
6    }
7  }
```

As explained earlier, the cast insertion mechanism for gradual effects inserts an effect cast before each effectful operation in the body of `map`. In this case, the only such expression is the application `f(x)` highlighted on line 5. The polymorphic effect system assigns `f` the maximal latent effect $\top$. Consequently, the effect cast is going to check at runtime whether the $\top$ effect can be performed. Statically, this means that the expression `map(l,print)` is valid in any context, since it has the unknown effect. Dynamically, if the list `l` is empty, no error is raised because line 5 is not reached. This is a slight improvement over the non-gradual version. On the other hand, if `l` is not empty, an effect error is raised even if the context allows for `@io` effects!

To address this issue, it suffices to consider that unannotated function arguments have *unknown* latent effects ¿, and not *maximal* effects $\top$ as defined in LPE. Let us go back to the `map` function again:

```
1  def map(l: List[Int], f: Int --@unknown--> Int):
2    List[Int] @unknown = {
3    l match {
4      case Nil => Nil
5      case x :: xs => f(x) :: map(xs, f)
6    }
7  }
```

Now, the application `f(x)` on line 5 is not preceded with a dynamic effect check, because `f(x)` statically produces the unknown effect, just like the whole body of the function.

To understand why this simple solution is sound, one must consider a call site. When checking `map(l,print)`, the gradual system now notices the mismatch between the latent effects of `print` (`@io`) and the expected latent effects of the `f` argument of `map` (`@unknown`). At this point, a *higher-order cast* is introduced [5]. Intuitively, this means that `print` is wrapped in a new function whose body performs the dynamic effect check for the statically-missing privileges (in this case, `@io`). Consequently, the use of `map` is sound and flexible: if `l` is empty, there is no runtime check at all. If `l` is not empty, the application `f(x)` first ensures that sufficient privileges are available to apply `f`.

Effect polymorphic casts. Unfortunately, the proper integration of gradual and polymorphic effect checking is not entirely solved yet. Consider that we cast the unannotated `map` function above to the effect-polymorphic type:[4]

$$(\texttt{l:List[Int]}, \texttt{f:Int} \xrightarrow{\texttt{@input}} \texttt{Int}) \xrightarrow[\texttt{f}]{\texttt{@pure}} \texttt{List[Int]}$$

The semantics of higher-order cast of TGE specifies that the casted `map` function first restricts the set of available privileges to `@pure`, and that the argument `f` is itself casted from the type $\texttt{Int} \xrightarrow{\texttt{@input}} \texttt{Int}$ to the type $\texttt{Int} \xrightarrow{\texttt{@unknown}} \texttt{Int}$.

[4] In the formalism of LPE, the specific effects of a polymorphic function are annotated above the arrow (here `@pure`), while the argument(s) on which the function is polymorphic (here `f`) are annotated below the arrow.

This cast, in turn, implies a runtime check for `@input` in the body of the argument function. Consequently, at runtime, even mapping a pure function fails: the check for `@input` occurs in a context that was restricted to `@pure`. This clearly conflicts with the semantics of effect polymorphism: the type above dictates that `map` ought to be pure, *except* when it applies `f`—where `@input` can be legally produced.

The solution we propose to resolve this tension is to recognize that whenever a function is casted to an effect-polymorphic type, we should skip the effect cast on the considered higher-order argument, and use its latent effects to dynamically restrict the context of privileges. In our example, this means not inserting a *check* for `@input`, but rather a *restriction* to `@input` when applying function `f`. Doing so is sound, because *at the call site* of the casted `map` function, the effect system has either statically established that the current context has the necessary privilege `@input`, or has inserted an effect cast to ensure that it is the case. In both cases, the casted function *cannot* produce more effects than what the current set of privileges allows when applying the casted function.

3.2 Customizable Effect System

We propose a domain-specific language, Effscript, to allow programmers to declaratively define effect disciplines. An *effect discipline* is composed of an *effect domain* and a number of *external effect specifications*. An effect domain is defined by a set of privileges, possibly type-parametrized, and a lattice that establishes subeffecting between effect privileges. External effect specifications (*effspecs* for short), permit both the external specification of effect annotations on function definitions as well as ascriptions. They also declare which expressions produce the effects of the domain.

To illustrate the use of Effscript, let us consider the refined IO domain presented in Figure 1. Suppose additionally that the programmer wants to track output only when printing `Person` objects. The complete discipline (domain and effspecs) is defined in Figure 2. The discipline is named `RichIO` (line 1); we now explain its definition in details.

Domain. The different effect privileges and the associated lattice can be seen in lines 2–8 of Figure 2. Lines 2-3 define the effect privileges of the domain. Notice that `@input` has a covariant type parameter, creating a subeffect relation according to the subtyping of the type parameter.

Lines 4–8 define the lattice. Line 5 (resp. 6) indicates which effect is to be considered the top (resp. bottom) of the lattice for the `RichIO` discipline. Line 7–8 are two subeffecting declarations, making both `@stdout` and `@stderr` subeffects of `@output`. When processing the discipline definition, Effscript checks that the lattice is well-formed (use of defined privileges, unicity of top and bottom elements, absence of cycles, consistent variance annotations).

External effect specifications. Lines 9–15 are the effspecs. An effspec is similar to a pointcut/advice pair in aspect-oriented programming [22, 24]: the first part, much like a pointcut, identifies code elements. Here, code elements are either function/method applications **`app`** or definitions **`def`**, identified using a simple pattern sub-language with wildcards and type constraints. The second part, much like a (domain-specific [14]) advice, specifies how to affect the identified code elements.

```
1  name: RichIO
2  effects: @noIo, @input[+T], @output, @stdout,
3           @stderr, @io
4  lattice:
5    top: @io
6    bottom: @noIo
7    @stdout <: @output
8    @stderr <: @output
9  effspecs:
10   app scala.Predef.read*:T prod @input[T]
11   app *.Predef.println(T<:Person) prod @output
12   app *.err.println(T<:Person) prod @stderr
13   app *.out.println(T<:Person) prod @stdout
14   def processStudents() ann @stdout @input[Int]
15   def loadStudents() ann @input[Student] within
16                                      controllers.*
```

Figure 2. Refined IO discipline using Effscript.

The **`prod`** operator means that the identified element *produces* the given effect. A type variable in the pattern is used to bind the corresponding type parameter in the produced effect. For instance, on line 10, the effspec states that any application of a method of the `Scala.predef` package that starts with `read` and returns a value of type `T` should be considered as producing an `@input[T]` effect. Lines 11-13 declare how the other effects are produced. Note the use of the type constraint `T<:Person` to express that only the printing of `Person` objects is of interest.

The **`ann`** operator means that the identified code element is annotated with the given effect. For a function definition, this corresponds to the latent effects of the function. For instance, line 14 specifies that the the latent effect of the `processStudents` function is `@stdout` and `@input[Int]`. This means that the function does not typecheck if it prints a `Person` object on the error output stream, or if it reads a non-integer input. Similarly, line 15 declares that it is valid for `loadStudents` to read `Student` objects only. Note that line 15 uses the **`within`** operator to annotate the function only if it is lexically inside any definition that belongs to the `controller` package. Using **`ann`** on an application instead of a definition results in an effect ascription, as in '`f(x) : @output`'.

4. A Theory of Gradual Polymorphic Effects

To precisely describe the semantics of the customizable gradual polymorphic effect system, we first describe a fusion of the theory of gradual effects of Bañados *et al* [5] (TGE) and the lightweight polymorphic effect system of Rytz and Odersky [30, 32] (LPE). As in both approaches, we focus on

a small core language, the lambda-calculus with a base type, extended with effect annotations. This section only contains highlights of the formal system. The complete formalization and soundness proof can be found in the companion technical report [37]. Section 5 describes the customizability of the system using Effscript. Section 6 discusses the Scala implementation.

4.1 Technical Background

The description of the gradual polymorphic effect system that follows heavily relies on both TGE and LPE. Therefore, this section strives to concisely describe the most important technicalities of both systems. The reader familiar with this prior work can safely jump directly to Section 4.2.

4.1.1 Gradual Effects

The theory of gradual effects is formulated as an extension to the generic effect system of Marino and Millstein [27]. The typing judgment has the form $\Phi; \Gamma \vdash e : T$, where Φ is a set of *effect privileges*, indicating what effects can be produced by a given expression e. To typecheck, an effectful operation requires the corresponding effect privilege to be in the privilege set Φ. Certain expressions grant privileges, thereby adjusting the privilege set Φ[5]. The system is generic in that the checking and adjusting of the available privilege set are parameters of the type system, defined as a $\mathbf{check}$ predicate and an $\mathbf{adjust}$ function.[6]

TGE introduces a new unknown effect, denoted ¿. One of the key insights of that work is to use abstract interpretation [9] 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 typing rules use a consistent privilege set Ξ instead of Φ in the context because of the unknown privilege. 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 $\Phi_2 \in \gamma(\Xi_2)$. Interestingly, the $\mathbf{check}$ and $\mathbf{adjust}$ functions of the generic effect framework can be automatically lifted to operate on consistent privilege sets, denoted $\widetilde{\mathbf{check}}$ and $\widetilde{\mathbf{adjust}}$, respectively. The exact definitions from [5] are not necessary to follow the description of our system.

Finally, as is standard in the formalization of gradual typing [34], the semantics of the language is given by translation to an internal language with dynamic checks. The evaluation judgment has the form $\Phi \vdash e \to e'$, 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.

4.1.2 Lightweight Polymorphic Effects

To provide a practical effect system for Scala, Rytz and Odersky propose Lightweight Polymorphic Effects [30, 32]. LPE is also based on the generic effect system of Marino and Millstein [27], albeit more loosely.

In particular, LPE is not formulated as a privilege checking system, but as an effect inference system. The type judgment of LPE has the form $\Gamma; \overline{x} \vdash e : T \,!\, \Phi$, where Φ is the set of inferred effects (output), instead of being part of the context (input). The $\overline{x}$ is called the *polymorphic context*, and is related to the main contribution of LPE, which is the mechanism for effect polymorphism.

A higher-order function (like `map`) is effect-polymorphic if its latent effects depend on the effects of some of its (higher-order) arguments. This is expressed by indicating *relative effects variables* on function types, meaning that each function declares a list of argument names $\overline{x}$ that contribute to its latent effects. For instance, the (curried) type of `map` is written $\texttt{List}[A] \xrightarrow{\bot} (f : A \xrightarrow{\top} B) \xrightarrow[f]{\bot} \texttt{List}[B]$, expressing that `map` is pure, save for the effects of its function argument f. This means that the actual latent effects of applying `map` are fully determined at *each call site*, once the type of the mapped function is known. In the typing judgment, the polymorphic context $\overline{x}$ keeps track of the variables on which the expression e is effect polymorphic. For the purpose of typechecking e, the application of functions contained in the polymorphic context is considered to be pure.

4.2 Language Syntax

We now turn to the formal description of the gradual polymorphic effect system. This section and the following describe the extensions and modifications to TGE that are necessary to integrate gradual effects with effect polymorphism, and subeffecting.

The syntax is shown in Figure 3. As in TGE, the language is parameterized on some finite set of privileges **Priv** for

[5] In this article, we use the terms "effect" and "privilege" interchangeably.

[6] More precisely, the $\mathbf{check}$ predicate is indexed by *check contexts* C, which represent the non-value expression forms of the language, and the $\mathbf{adjust}$ function is indexed by *adjust contexts* A, which represent the immediate context around a given subexpression [5, 27]. For instance, in a language with only function application, there is one check context, and two adjust contexts, which correspond to evaluating each sub-expressions. For simplicity in the notation, we simply refer to $\mathbf{check}$ and $\mathbf{adjust}$, leaving their contextual indexes implicit.

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

$$\begin{array}{rcll} v & ::= & \texttt{unit} \mid (\lambda x\colon T\,.\,e)^{T;\Xi;\overline{x}} & \text{Values} \\ e & ::= & x \mid v \mid e\ e \mid e :: \Xi & \text{Terms} \\ T & ::= & \texttt{Unit} \mid (x\colon T) \xrightarrow[\overline{x}]{\Xi} T & \text{Types} \end{array}$$

Figure 3. Language syntax

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 ¿.

A value can either be **`unit`** or a function. The main difference with TGE is that functions are fully annotated[7], including the type of the argument T_1, the return type T_2, the latent (consistent) privilege set Ξ, and the relative effect variables $\overline{x}$. A term e can be a variable x, a value v, an application $e\ e$, or an effect ascription $e :: \Xi$. A type is either `Unit` or a function type $(x\colon T) \xrightarrow[\overline{x}]{\Xi} T$. Although functions have only one argument, the relative effect variables $\overline{x}$ can include variables defined in the surrounding lexical context.

For instance, in a context Γ where f is defined, a function that takes a function g as argument, performs some output, and applies both f and g, can be defined as follows:

$$(\lambda g\colon \texttt{Unit} \xrightarrow{\top} \texttt{Unit}\ .\ ...)^{\texttt{Unit};\{\texttt{@output}\};\{f,g\}}$$

4.3 Type System

The complete type system is presented in Figure 4; the changes with respect to TGE are highlighted in gray. In particular, the type judgement $\Xi;\Gamma;\overline{x} \vdash e\colon T$ includes the additional polymorphic context $\overline{x}$. Also, the rules rely on two specific notions: *consistent subcontainment*, an extension of consistent containment that supports subeffecting, and a new definition of *consistent subtyping* that additionally takes polymorphic contexts into account.

Consistent subcontainment. Subeffecting yields a more flexible notion of set containment, called *subcontainment*:

Definition 1 (Subcontainment)**.** *Φ_1 is* subcontained *in Φ_2, notation $\Phi_1 \subseteq: \Phi_2$, if and only if $\forall \phi_1 \in \Phi_1, \phi_1 \in \Phi_2 \vee \exists \phi_2 \in \Phi_2$ such that $\phi_1 <: \phi_2$.*

Based on this notion of subcontainment, we can define *consistent subcontainment* as an extension of consistent containment (recall Section 4.1.1) that deals with subeffecting:

Definition 2 (Consistent Subcontainment)**.** *Ξ_1 is* consistently subcontained *in Ξ_2, notation $\Xi_1 \sqsubseteq_\sim: \Xi_2$, if and only if $\Phi_1 \subseteq: \Phi_2$ for some $\Phi_1 \in \gamma(\Xi_1)$ and $\Phi_2 \in \gamma(\Xi_2)$.*

[7] We further discuss annotations in Section 6.1.

$$\boxed{\Xi;\Gamma;\overline{x} \vdash e\colon T} \qquad \textsc{Var}\ \frac{\Gamma(x) = T}{\Xi;\Gamma;\overline{x} \vdash x\colon T}$$

$$\textsc{Fn}\ \frac{\Xi_1;\Gamma, x\colon T_1;\overline{x_1} \vdash e\colon T' \qquad T' \lesssim: T_2}{\Xi;\Gamma;\overline{x} \vdash (\lambda x\colon T_1\,.\,e)^{T_2;\Xi_1;\overline{x_1}} : (x\colon T_1) \xrightarrow[\overline{x_1}]{\Xi_1} T_2}$$

$$\textsc{Eff}\ \frac{\Xi_1;\Gamma;\overline{x} \vdash e\colon T \qquad \Xi_1 \sqsubseteq_\sim: \Xi}{\Xi;\Gamma;\overline{x} \vdash (e :: \Xi_1)\colon T}$$

$$\textsc{App}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}(\Xi);\Gamma;\overline{x} \vdash e_1\colon (y\colon T_1) \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \widetilde{\textbf{adjust}}(\Xi);\Gamma;\overline{x} \vdash e_2\colon T_2 \\ \Xi_1{}' = \Xi_1 \cup (\cup_{f \in (\overline{y} \setminus \overline{x})} \mathit{latent}_{\Gamma;\overline{x}}((\Gamma, y\colon T_2)(f))) \\ e_1 \notin \overline{x} \qquad \Xi_1{}' \sqsubseteq_\sim: \Xi \qquad T_2 \lesssim: T_1 \qquad \widetilde{\textbf{check}}(\Xi)\end{array}}{\Xi;\Gamma;\overline{x} \vdash e_1\ e_2\colon T_3}$$

$$\textsc{AppP}\ \frac{\begin{array}{c}\Gamma(f) = (y\colon T_1) \xrightarrow[\overline{y}]{\Xi_1} T_3 \qquad \widetilde{\textbf{adjust}}(\Xi);\Gamma;\overline{x} \vdash e_2\colon T_2 \\ f \in \overline{x} \qquad T_2 \lesssim: T_1 \qquad \widetilde{\textbf{check}}(\Xi)\end{array}}{\Xi;\Gamma;\overline{x} \vdash f\ e_2\colon T_3}$$

Figure 4. Gradual polymorphic effects: type rules.

Consistent subtyping. Consistent subtyping from TGE extends subtyping to deal with consistent privilege sets. Modifying consistent subtyping from TGE to use consistent subcontainment is enough for dealing with subeffecting, but is not sufficient for polymorphism: we need to account for the relative effect variables on function types, as in LPE. The full definition of subtyping for effect polymorphism is quite involved [30]. Consistent subtyping $\lesssim:$ reuses the definition of subtyping from Rytz, except for the fact that it relies on consistent subcontainment, instead of the simple containment relation used by Rytz to compare effect sets (see [37] for details).

Typing rules. Rule [Var] is self explanatory. Rule [Fn] typechecks the body of the function using the annotated privilege set Ξ_1 and relative effect variables $\overline{x_1}$, and verifies that the type of the body T' is a consistent subtype of the annotated return type T_2.

To type an effect ascription (rule [Eff]), the ascribed privilege set is used to typecheck the inner expression. This rule is the same as in TGE save for the polymorphic context and the fact that is uses consistent subcontainment to check that the ascribed privilege set is valid in the current context.

Rule [App] is an adaptation of the corresponding TGE typing rule to support relative effects. The sub-expressions

e_1 and e_2 are typed using adjusted privilege sets (according to each domain). $\widetilde{\mathbf{check}}$ verifies that the application is allowed with the given permissions Ξ. A subtlety is that if the invoked function is effect-polymorphic, its latent effects are not only Ξ_1, but also include the latent effects of the relative effect variables of the functions in $\overline{y}$ that are not already present in the polymorphic context $\overline{x}$.

These additional latent effects are computed by the auxiliary function $\mathit{latent}_{\Gamma;\overline{x}}(T)$ defined in [30]. The function needs access to both the type environment Γ and the polymorphic context $\overline{x}$ to lookup the types of the relative effect variables. An extra subtlety is that the type of each f in $\overline{y}\backslash\overline{x}$ is obtained in an environment in which the argument y has type T_2, not T_1. This is to account for effect polymorphism: the actual latent effects of the argument come from e_2.

Rule [AppP] is a new rule for the application of functions that are the parameter of an enclosing effect-polymorphic function (*i.e.* $f \in \overline{x}$). The difference between [AppP] and [App] is very subtle: the typing rule [AppP] does not need to check if the latent effects of the function being applied are consistently subcontained in the set of privileges of the enclosing application. The reason is that in [AppP] the application is being polymorphic on f, meaning that the application is allowed to produce any effect that f may produce.

4.4 Extension of the Translation Rules

The source language supports unknown privilege annotations, therefore runtime checks must be introduced. To introduce runtime checks, the source language is translated into an internal language, which makes runtime checks explicit. The rules are similar to TGE except for the support for effect polymorphism, subeffecting, and the way we deal with higher-order casts. Figure 5 presents the most significant translation rules, *i.e.* for function application.

Rule [TApp] describes the non-polymorphic function application. There are two main differences compared to [App]. First, a runtime check may be introduced using *insert-has?*, to determine whether the statically-missing privileges in Ξ to perform the application are available at runtime. This privilege set Φ is obtained using the metafunction Δ defined in [5], which computes the minimal set of additional privileges needed to safely pass the $\widetilde{\mathbf{check}}$ verification. The metafunction *insert-has?* inserts a dynamic check for privileges only if the privilege set Φ is not empty.

Rule [TAppP] is the transformation rule for applications of functions that are the parameter of an enclosing effect-polymorphic function. It is very similar to [TApp] save for the fact that the application expression is a variable f, so there is no recursive translation.

Cast insertion. Both rules [TApp] and [TAppP] use the cast insertion metafunction $\langle\!\langle\cdot\rangle\!\rangle^{c}_{\Gamma}$ to cast the application expression to the appropriate type. An inserted cast has the form $\langle T_2 \Leftarrow T_1\rangle^{c}_{\Gamma}$. As a standard optimization, a cast is only inserted if the casted expression is not a static subtype of the target type. The c and Γ annotations in both cases are new compared to TGE. The boolean variable c is used to indicate whether the cast must eventually include the dynamic effect check `has` or not: c is true if an application is not polymorphic [TApp], and false if the application is polymorphic [TAppP]. The annotated type environment Γ is technically necessary to resolve latent effects in casts.

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

$$\text{TApp}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}(\Xi);\Gamma;\overline{x} \vdash e_1 \Rightarrow e_1' : (y:T_1) \xrightarrow[\overline{y}]{\Xi_1} T_3 \qquad \widetilde{\mathbf{adjust}}(\Xi);\Gamma;\overline{x} \vdash e_2 \Rightarrow e_2' : T_2 \\ \Xi_1' = \Xi_1 \cup (\cup_{f\in(\overline{y}\backslash\overline{x})} \mathit{latent}_{\Gamma;\overline{x}}((\Gamma, y:T_2)(f))) \\ \Xi_1' \sqsubseteq_{\sim} \Xi \qquad T_2 \lesssim T_1 \\ e_1'' = \langle\!\langle (y:T_2) \xrightarrow{\Xi} T_3 \Leftarrow (y:T_1) \xrightarrow[\overline{y}]{\Xi_1} T_3 \rangle\!\rangle^{\mathtt{true}}_{\Gamma} e_1' \\ e_1 \notin \overline{x} \qquad \widetilde{\mathbf{check}}(\Xi) \qquad \Phi = \Delta(\Xi)\end{array}}{\Xi;\Gamma;\overline{x} \vdash e_1\ e_2 \Rightarrow \mathit{insert\text{-}has?}(\Phi, e_1''\ e_2') : T_3}$$

$$\text{TAppP}\ \frac{\begin{array}{c}\Gamma(f) = (y:T_1) \xrightarrow[\overline{y}]{\Xi_1} T_3 \qquad \widetilde{\mathbf{adjust}}(\Xi);\Gamma;\overline{x} \vdash e_2 \Rightarrow e_2' : T_2 \\ e_f = \langle\!\langle (y:T_2) \xrightarrow{\Xi} T_3 \Leftarrow (y:T_1) \xrightarrow[\overline{y}]{\Xi_1} T_3 \rangle\!\rangle^{\mathtt{false}}_{\Gamma} f \\ f \in \overline{x} \qquad T_2 \lesssim T_1 \qquad \widetilde{\mathbf{check}}(\Xi) \qquad \Phi = \Delta(\Xi)\end{array}}{\Xi;\Gamma;\overline{x} \vdash f\ e_2 \Rightarrow \mathit{insert\text{-}has?}(\Phi, e_f\ e_2') : T_3}$$

Figure 5. Translation to the internal language: rules for application.

Higher-order effect casts. In the theory of gradual effects [5], translation to the internal language introduces higher-order casts, whose semantics is then given in the dynamic semantics:

$$\begin{array}{l}\langle T_{21} \xrightarrow{\Xi_2} T_{22} \Leftarrow T_{11} \xrightarrow{\Xi_1} T_{12}\rangle\ (\lambda x{:}\,T_{11}\,.\,e) \rightarrow \\ \quad (\lambda x{:}\,T_{21}\,.\,\langle T_{22} \Leftarrow T_{12}\rangle \mathtt{restrict}\ \Xi_2\ \mathtt{has}\ (|\Xi_1|\backslash|\Xi_2|)) \\ \qquad\qquad [(\langle T_{11} \Leftarrow T_{21}\rangle x)/x]e)\end{array}$$

That is, a higher-order cast reduces to a function wrapper that `restrict`s the current privilege set to Ξ_2, and then checks that the context `has` the minimal privilege set not already accounted for statically. Note that a cast on the argument is also inserted, which may turn out to be higher-order as well. This semantics requires the runtime system to have dedicated support for higher-order casts.

We adopt instead an equivalent formulation in which higher-order casts are given semantics directly during the translation phase: a cast is statically expanded to the corresponding literal function wrapper. The downside of this approach is that it forces us to introduce an internal application operator $\bullet$ to apply wrappers without interfering with type checking. However, this operator is important for type-

checking only, and its existence at runtime is necessary only to formally prove type soundness. A consequence of soundness is that it can be erased, and all applications are dealt with in the same way. Consequently, we avoid modifying the runtime semantics of the language.

As explained in Section 3.1, a key challenge in integrating gradual and polymorphic effect checking is the ability to cast a function to an effect-polymorphic type. The solution is to not insert a **has** check in the function wrapper of an argument on which the target function type is polymorphic, and to allow a richer context of privileges using **restrict**. Therefore, the semantics of higher-order effect casts depends on whether the cast is monomorphic or polymorphic.

Monomorphic effect casts. A monomorphic effect cast is defined as follows:[8]

$$\begin{array}{l}\langle (x_2 : T_{21}) \xrightarrow[\overline{x_2}]{\Xi_2} T_{22} \Leftarrow (x_2 : T_{11}) \xrightarrow[\overline{x_1}]{\Xi_1} T_{12} \rangle_\Gamma^{\texttt{true}}\ f = \\ \quad (\lambda x_2 \colon T_{21}\ .\ \langle\!\langle T_{22} \Leftarrow T_{12} \rangle\!\rangle_\Gamma^{\texttt{true}} \\ \qquad \textbf{restrict}\ \Xi_2 \cup (\cup_{x \in (\overline{x_2})} \mathit{latent}_{\Gamma;\emptyset}((\Gamma, x_2 \colon T_{21})(x))) \\ \qquad \textbf{has}\ |\Xi_1 \cup (\cup_{x \in (\overline{x_1} \setminus \overline{x_2})} \mathit{latent}_{\Gamma;\emptyset}((\Gamma, x_2 \colon T_{21})(x)))| \setminus |\Xi_2| \\ \qquad\quad f \bullet (\langle\!\langle T_{11} \Leftarrow T_{21} \rangle\!\rangle_\Gamma^{x_2 \notin \overline{x_2}}\ x_2))^{T_{21};\Xi_2;\overline{x_2}}\end{array}$$

The general scheme is overall the same as in TGE: the cast is compiled into a new function that performs a **restrict/has** combination, casting both the argument and the return value. Compared to TGE, the cast insertion function $\langle\!\langle \cdot \rangle\!\rangle_\Gamma^c$ is used in order to recursively compile these inner casts. The cast on the return type always inserts a dynamic check (c is true) because there is no polymorphism on return values. For the argument cast, c is true only if the target type of the cast is not polymorphic in its argument x_2, *i.e.* $x_2 \notin \overline{x_2}$. Also, note that while TGE uses direct substitution to silently apply the casted function (a trick that is only applicable because casts are given specific runtime semantics), we resort to a primitive application operator $\bullet$, which does not cause additional effect checks. The most interesting part of the definition is that inserted **restrict** must include the latent effects of the relative effect variables of the target type, because they represent the maximal privilege set that x_2 *may* produce. Also, the inserted **has** must check for the latent effects of the relative effects variables of $\overline{x_1} \setminus \overline{x_2}$, because they represent the maximal privilege set that $\overline{x_1}$ *may* produce and that $\overline{x_2}$ does not produce.

Polymorphic effect casts. A polymorphic effect cast is defined as follows:

$$\begin{array}{l}\langle (x_2 : T_{21}) \xrightarrow[\overline{x_2}]{\Xi_2} T_{22} \Leftarrow (x_1 : T_{11}) \xrightarrow[\overline{x_1}]{\Xi_1} T_{12} \rangle_\Gamma^{\texttt{false}}\ f = \\ \quad (\lambda x_2 \colon T_{21}\ .\ \langle\!\langle T_{22} \Leftarrow T_{12} \rangle\!\rangle_\Gamma^{\texttt{true}} \\ \qquad \textbf{restrict}\ \Xi_2 \cup (\cup_{x \in (\overline{x_2})} \mathit{latent}_{\Gamma;\emptyset}((\Gamma, x_2 \colon T_{21})(x))) \\ \qquad\quad f \bullet_f (\langle\!\langle T_{11} \Leftarrow T_{21} \rangle\!\rangle_\Gamma^{x_2 \notin \overline{x_2}}\ x_2))^{T_{21};\Xi_2;\overline{x_2}}\end{array}$$

The most significant difference with the monomorphic case is that no **has** check is introduced. This adaptation of **restrict/has** corresponds to the flexibility of effect polymorphism: applying a function on which the expression is polymorphic is considered to not produce any effect (no **has**), but the permitted effects must be bounded by the declared latent effects of that function (with **restrict**). Note also that the primitive application operator used to apply f is annotated, $\bullet_f$, in order to keep track of the fact that the latent effects of f need not be considered.

As an illustration, Appendix A provides a step-by-step derivation of (a simplified version of) the `map` example of Section 3.1.

4.5 Soundness

Type soundness of the gradual polymorphic effect system is proven by proving soundness of the internal language using a standard preservation and progress argument. We then prove that the translation from the source language to the internal language preserves typing, thereby establishing type soundness for the gradual language. The complete definition and soundness proof is available in the companion technical report [37].

5. Customizing the Effect System

This section briefly discusses how Effscript, the domain-specific language for specifying effect disciplines, is defined. The syntax of Effscript is fairly straightforward (Appendix B). In particular, the syntax of external effect specifications (*effspecs*) is directly based on pointcuts in aspect-oriented languages, in particular in the static pointcut designators of AspectJ such as **call** and **within** [22]. We have settled for a simple design that is expressive enough to cover the needs we encountered in practice so far.

We describe the semantics of Effscript through extensions of the semantics of the gradual polymorphic effect systems of the previous section. In this paper, we focus on the semantics of annotation specifications (with **ann**) and production specifications (with **prod**). In short, annotation specifications are realized through a pass of code transformation *prior* to effect checking, while production specifications are dealt with directly in the effect system.

Similarly to the class table in Featherweight Java [20], which holds the information about class definitions and is implicitly available in the typing rules, we introduce an effspec context, which is used by matching functions to determine whether an effspec applies to the considered expression. We do not explicitly pass this context around.

Annotation specifications. The **ann** operator is given semantics through a source-to-source transformation that inserts effect ascriptions where needed. To determine whether

[8] The definition assumes that the casted expression is a variable f. This is because in case the expression is an arbitrary e, the cast insertion function introduces an internal function application to ensure that e is evaluated first before the wrapper is created. This synthetic application is realized with a primitive application operator that does not cause additional effect checks.

an application or definition must be updated, the matching function $\mathbf{match}_{\mathtt{a}}$ relies on the type information. Therefore, the source-to-source translation is defined as a judgment of the form $\Gamma \vdash e \rightsquigarrow e' : T$. Here is the transformation rule for an application that is matched by at least one effspec:

$$\text{AApp-m}\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : T_1 \longrightarrow T_3 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : T_2 \quad \Xi = \mathbf{match}_{\mathtt{a}}(e_1\ e_2, T_1 \longrightarrow T_3, T_2) \quad \Xi \neq \emptyset}{\Gamma \vdash e_1\ e_2 \rightsquigarrow (e_1'\ e_2' :: \Xi) : T_3}$$

$\mathbf{match}_{\mathtt{a}}$ can exploit the syntactic information of e_1 and e_2 as well as their respective types in order to determine if some effspecs match the application expression. If so, it returns a consistent privilege set Ξ that is the union of all the declared annotations. Ξ is then used to insert the ascription. Note that effects are not taken into account at this early stage. The simple typing phase is necessary to provide $\mathbf{match}_{\mathtt{a}}$ with the type information of e_1 and e_2. The type-and-effect system defined previously applies on the transformed code in which all annotations have been introduced.

Production specifications. The `prod` operator updates the effects produced by an application or the latent effects of a function definition. The semantics of production specifications is integrated in the effect system. Here is the modified rule for an application that is matched by at least one effspec:

$$\text{PApp-m}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}(\Xi);\Gamma;\overline{x} \vdash e_1 : (y : T_1) \xrightarrow[\overline{y}]{\Xi_1} T_3 \\ \widetilde{\mathbf{adjust}}(\Xi);\Gamma;\overline{x} \vdash e_2 : T_2 \\ \Xi_1' = \Xi_1 \cup (\cup_{f \in (\overline{y} \setminus \overline{x})} latent((\Gamma, y : T_2)(f))) \\ \Xi_1' \sqsubseteq_{\sim} \Xi \qquad T_2 \lesssim : T_1 \\ \Xi' = \mathbf{match}_{\mathtt{p}}(e_1\ e_2, T_1 \longrightarrow T_3, T_2) \\ \Xi' \neq \emptyset \qquad \Xi' \sqsubseteq_{\sim} \Xi\end{array}}{\Xi;\Gamma;\overline{x} \vdash e_1\ e_2 : T_3}$$

As for annotation specifications, $\mathbf{match}_{\mathtt{p}}$ returns the union of the produced effects of all matching effspecs, or the empty set if none match. If some effspecs match, then the only check that is realized is that the current context Ξ includes sufficient privileges for the new produced effects Ξ'.

Limitations of effspecs. All aspect-oriented extensions of languages with first-class functions have to face the challenge of denoting functions in that may be aliased with pointcuts [10, 13, 36]: for instance, detecting the call to `print` in `val f=print; f("foo");`. Dynamic languages such as AspectScheme and AspectScript typically resort to function pointer equality at runtime [13, 36]; the alternative is to stick to compile-time pointcut matching, and limit pointcuts to denote named function definitions without dealing with aliases (*e.g.* AspectML [10]). To preserve static matching, we adopt the latter approach and hence do not match application of aliases to first-class functions.

Finally, the expressiveness of effspecs is rather limited so far. For instance, we cannot refine the allowed privilege set on the argument of an application, such as:

```
app spawn(e) ann (e) as @pure
```

Extending the expressiveness of effspecs is an interesting venue for future work.

6. Implementation in Scala

The customized gradual effect system for Scala is implemented as a compiler plugin for `scalac`, and an external DSL, Effscript. Effscript is implemented simply using Scala parsing combinators to process `.eff` files, and outputs Scala source files with class definitions of the defined effect discipline, which are used by the compiler plugin. The compiler plugin itself is composed of two sub-plugins to implement bidirectional effect checking as explained in Section 6.1 below. Finally, we discuss how we support the semantics of the gradual effect system without modifying the runtime environment (JVM) in Section 6.2.

6.1 A Bidirectional Effect System

The original system of Bañados *et al* does not require lambda abstractions to be annotated with their latent effects; instead, the latent effects are (non-deterministically) chosen. Consider the abstraction type rule from [5]:

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

The function type uses Ξ_1 as the latent effect sets, which corresponds to any set that is sufficient to typecheck the body e. This non-determinism is not satisfactory for implementing the system, as it leaves the question of how to determine Ξ_1 open. This is why the type system we have presented is based on fully annotated functions, including their latent effects. The downside is that requiring fully-annotated functions can be impractical.

To resolve this tension, we have implemented a *bidirectional* effect system, similar to the bidirectional type checking approach of Pierce and Turner [29]. The system has two working modes: a checking mode, in which the permitted effects are given in the typing context, and an inference mode, in which latent effects are an output of the type system. Effect ascriptions and annotations trigger mode switches. In particular, switching to inference mode allows the system to infer the minimal valid effect set Ξ_1 of a function definition.

To implement the bidirectional effect system, the system works in two phases. The first phase takes a source program with possibly missing annotations and produces a fully-effect annotated program, using effect inference. Note that this first phase reuses the effect plugin developed by Rytz, which performs polymorphic effect inference, modified to take into account unknown effects and the Effscript customizations. The second phase applies the effect checking system on the fully-annotated code, together with the

```
1  def f(x: Int): Unit @unknown = {
2    (/* body */) : @input
3  }
4  f(1) : @output
5  /*****************************************/
6  def f(x: Int): Unit = {
7    RuntimePrivileges.has(List(input()));
8    /* body */
9  }
10 RuntimePrivileges.restrict(List(output())){
11   f(1)
12 }
```

Figure 6. (top) Sample program, which typechecks but fails at runtime. (bottom) Sample program after transformation.

translation for inserting **has** and **restrict** where needed, exactly as described in Section 4.4. This phase is implemented as a separate sub-plugin.

The translation to fully-annotated code is straightforward. The judgement $\Gamma; \overline{x} \vdash e \Rightarrow e' : T \,!\, \Xi$ describes that the expression e is translated to the (fully-annotated) expression e', of type T and produces effects Ξ. The only interesting rule is the rule for the unannotated abstraction, given below:

$$\frac{\Gamma, x : T_1; \emptyset \vdash e \Rightarrow e' : T \,!\, \Xi}{\Gamma; \overline{x} \vdash (\lambda x : T_1 \,.\, e) \Rightarrow (\lambda x : T_1 \,.\, e)^{T;\Xi;\emptyset} : (x : T_1) \xrightarrow[\emptyset]{\Xi} T \,!\, \emptyset}$$

The rule infers the return type T and the effects produced by the function body Ξ. This information is used to translate the lambda to a fully-annotated version. Note that, according to the semantics of LPE, each function (even nested) should explicitly declare its relative effect variables. In this case, the unannotated lambda does not include such information, so the polymorphic context is empty, and so is the relative effects of the translated function.

6.2 Runtime Effect Checking

The gradual effect system is defined with a non-standard runtime semantics to support checking effects at runtime [5]. The semantics include an extra context information to track the current privilege set, and there are dedicated rules to deal with **has**, **restrict**, and higher-order casts. A direct implementation of this semantics would imply modifying the JVM, which is not an option for the adoption of the effect system by mainstream Scala programmers. Instead, we transform Scala source code to an equivalent Scala code that makes use of a small runtime library to track and check effects at runtime.

Figure 6(top) shows a small program, which typechecks, but fails at runtime because function `f` is trying to produce `@input` in a context where only `@output` is allowed. The compiler plugin transforms the code into that of Figure 6(bottom).

The tracking of effect privileges in the context is implementing using a dynamically-scoped variable (called *dynamic variable* in Scala), named `RuntimePrivileges`. This object mainly provides two methods: **has** (line 7) to check that the list of effects given as argument is compatible with the current set of privileges, and **restrict** (line 10) to adjust the set of privileges for the dynamic extent of its body (the last argument delimited by curly braces).

The calls to **has** and **restrict** are inserted as needed based on the (static) comparison of the required privilege set and the privilege set available in the context. For instance, the transformation shown in Figure 6 assumes that the context of the `f(1)` call statically includes `@output`, hence no **has** is inserted on line 10.

Representing effects. Note that the code after transformation uses a runtime representation of effects: `@input` and `@output` are represented as (lowercase-named) classes `input` and `output`, and privilege sets are represented as `List`s, as shown in lines 2 and 7. As we will see in Section 7.3, the choice of more efficient runtime representation for both effects and privilege sets can significantly affect the observed overhead of runtime effect checking. Also, representing effects with type parameters is particularly challenging due to the fact that the JVM does not support runtime type parameters. Consequently, we have to resort to strings and reflection to compute subeffecting at runtime, which is costly (see Section 7.3).

Higher-order casts. While the semantics of TGE treat casts through dedicated reduction rules, here we take advantage of the translation-based realization of casts described in Section 4.4. This equivalent formulation has the advantage of not requiring new evaluation semantics: higher-order casts are handled simply with anonymous functions, **restrict**, and **has** operations.

7. Experience and Validation

We now report on different efforts to experiment with and validate the implementation of the customizable gradual effect system for Scala. We first report on the use of Effscript to impose architectural constraints in web applications developed in the popular Play framework. Second, we discuss how we can exploit gradual checking and external effect specifications to integrate existing libraries in a system with effects. Finally, we report on some benchmarks that assess the cost of runtime effect checking, and the pay-as-you-go motto of gradual typing.

7.1 Architectural Constraints in Play

Play is a popular web framework for Scala and Java [39]. A typical Play application follows the MVC architectural pattern. The MVC pattern splits applications into three layers: the model layer, the view layer and the controller layer. The model components define the data entities on which the application operates, and usually relies on a persistence storage mechanism to store the data, such as a database. The

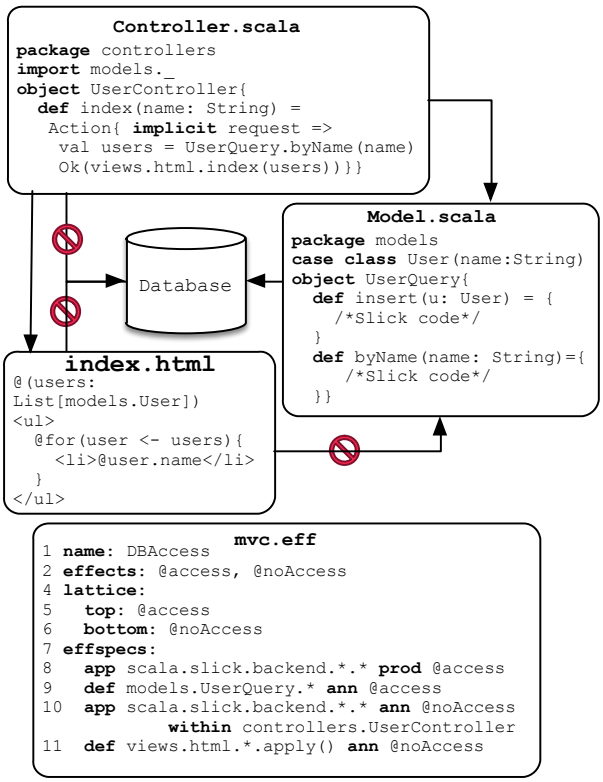

Figure 7. Enforcing the MVC pattern in Play with Effscript.

view components render the model into a user interface. The controller components respond to events and may access the models.

The MVC pattern mandates a number architectural constraints. Most importantly, only models should be able to access the database directly. Controllers can access the database only through models, not directly. Views should not be able to access the database, either directly or through calling methods on model objects that access the database (they are only allowed to read properties of model objects). Of course, the standard Scala type system falls short when it comes to ensuring that a Play application really follows these architectural constraints. We can use Effscript to declaratively specify an effect discipline for these MVC constraints for a given Play application, so that the type system helps programmers respect the pattern.

Figure 7 shows a `UserController` controller that queries all users in the database by calling the `byName` method of the `UserQuery` model, which uses the Slick library [38] for database access. The result is rendered by the `index.html` template. The figure shows the interactions that should be prevented by the effect system.

The Effscript file `mvc.eff` on Figure 7, lines 1–6, defines the custom effect discipline to enforce the MVC pattern in the Play application. It declares two effects, `@access` and `@noAccess`, to represent database access privileges. With this effect domain specification, one can either use effect annotations explicitly in the application code, or use effspecs (as in lines 7–11). We discuss both approaches below.

Explicit effect annotations. Directly using the effect discipline requires the model to be updated in order to declare where effects are produced:

```
1  package models
2  case class User(id: Long, name: String,
3          parentId: Long)
4  object UserQuery{
5    val all = TableQuery[UserTable]
6    def insert(u: User): Unit @access = {
7      all += u
8    }
9    def byName(name: String):
10     List[User] @access = {
11       all.filter(_.name===name).list
12   }
13 }
```

Line 6 and 9 declare that those methods produce the `@access` effect. In addition, to prevent the controller or the view from access the database directly, we need to use explicit effect ascriptions, for instance:

```
1  package controllers
2  import models._
3  object UserController{
4    def index(name: String) = DBAction {
5      implicit rs =>
6        val users = UserQuery.byName(name)
7        UserQuery.all +=
8                    User("John", 1) : @noAccess
9        Ok(views.html.index(users))
10   }
11 }
```

One could argue that instead of introducing the ascription, it would be preferable to avoid performing a direct call to the database in the first place. However, it can be hard to avoid writing these explicit ascriptions without using effects. For instance, Play uses Twirl [9] as a template engine, which transform views to objects. As of today, Twirl does not support annotation of views. Therefore we cannot simply declare `index.html` to be pure and have to use explicit ascriptions as above on each effectful operation of a view.

Finally, unless we are willing to manually introduce effect annotations and recompile the whole Slick library and its dependencies, we have to assume that Slick functions always potentially produce maximal effects, which is impractical.

Effspecs to the rescue. Effspecs elegantly solve all the problems described above. We can declaratively specify the effects allowed in the views, as well as which effects are produced where, without having to alter the source code:

```
6  ...
7  effspecs:
8    app scala.slick.backend.*.* prod @access
9    def models.UserQuery.* ann @access
10   app scala.slick.backend.*.* ann @noAccess
         within controllers.UserController
11   def views.html.*.apply() ann @noAccess
```

[9] `https://github.com/playframework/twirl`

Line 8 declares that every method of the Slick library produces `@access`. Line 9 grants the `UserQuery` model the privilege to access the database, by adding `@access` to the latent effects of all its methods. Line 10 declares that `UserController` is not allowed to directly access the database through the Slick library, but it can still access the database through methods of the model. Finally line 11 declares that views cannot access the database at all: the latent effect of every function of the `views.html` package must be pure. This means that directly calling methods from the Slick backend or methods from models that access the database is rejected by the effect system.

Interestingly, even implicit calls to effectful operations in the views can be detected and prevented. Suppose a method `getChildren` of class `User` that uses a method of `UserQuery`. A simplified view that shows the information of a user could be declared as follows:

```
1  @(user: models.User)
2  <p>Name: @user.name</p>
3  <p>Children: @user.getChildren.mkString(",")</p>
```

Line 3 accesses the database implicitly through the method `getChildren`, hence violating the MVC restriction. Thanks to the defined effspecs and effect propagation, the effect system infers that `getChildren` produces `@access`, and therefore it is rejected statically.

Refining the restriction to certain tables. We can further refine the effect discipline by forbidding `UserController` to insert in the database new model objects other than users. This refinement is formulated in Effscript by extending the definition of Figure 7 as follows:

```
name: DBAccess
  effects:  ..., @insert[+T]
  lattice:
    ...
    insert[T] <: @access
  effspecs:
    ...
    app models.*.insert(T) prod @insert[T]
    app models.*.insert[T] ann @insert[User]
           within controllers.UserController
```

To track insertions of model objects of type `T` in the database, a new (covariant) effect privilege `@insert[+T]` is declared. In the lattice section, we declare that `@insert[+T]` is a subeffect of `@access`. Finally, two new effspecs are added: the first specifes that methods called `insert` of package `model` produce `@insert[+T]` binding the type of the method argument `T` to the type parameter of the produced privilege. The second effspec declares that whenever a `UserController` invokes such an `insert` method, the application expression is wrapped with an effect ascription `insertUser`, thereby ensuring that only insertion of `User` objects are allowed.

For instance, suppose that a `BlogArticle` is accidentally inserted in the `UserController`:

```
1  package controllers
2  import models._
3  object UserController{
4    def insert(name: String) = DBAction {
5      implicit rs =>
6        val article = BlogArticle(...)
7        BlogArticleQuery.insert(article)
8         Ok(views.html.flash("success"))
9    }
10 }
```

The effect system reports an error informing that line 7 is produces `insert[BlogArticle]` whereas the only allowed effect is `insert[User]`.

7.2 Interfacing with Existing Libraries

Integrating external libraries in a system that uses effect disciplines is challenging. One can either (over conservatively) assume that external functions can produce any effect ($\top$), or unsafely assume that they are pure ($\bot$). In both cases, the fact that the decision is a global language default is too rigid. A more flexible option consists in generating stub files for external libraries and manually annotate each external function with its intended types—such an approach is supported for instance in the Checker framework for pluggable type systems [11], or to annotate existing JavaScript libraries in TypeScript [7].

The system we propose in this work enhances the situation on two fronts. First, by supporting *gradual* effect checking, there is an interesting alternative default option to the $\top/\bot$ dilema. By considering unannotated functions to produce *unknown* effects, one resorts to dynamic checking for third party libraries. Technically, a limitation is that the gradual checking approach of Bañados *et al*, adopted here, works by translating code to insert dynamic checks (**has**) and context adjustments (**restrict**). Therefore, the code of the external library should be available for processing.[10]

Interestingly, when external function *definitions* themselves cannot be altered, effspecs in Effscript provide another viable alternative: an effspecs can denote all *applications* of the external function, just as we did in the MVC example (Figure 7) to externally specify that calling Slick backend methods produces the `@access` effect.

This leads us to the other advantage of our approach for dealing with external libraries: the pattern matching language (akin to pointcuts) allows us to attach default effect annotations to several library functions at once, without having to go through each function manually, as is necessary with the stubs approach.

For instance, for the MVC example of Section 7.1, we include the following effspecs in an initial global Effscript file, to prevent effect tracking of external library functions in which we are not interested:

[10] The fact that our implementation works as a source-level modification instead of bytecode-level modification is a technical detail; we could exploit bytecode manipulation tools instead. But this does not address the issue of foreign function interfaces, or libraries for which the bytecode cannot be altered for copyright reasons.

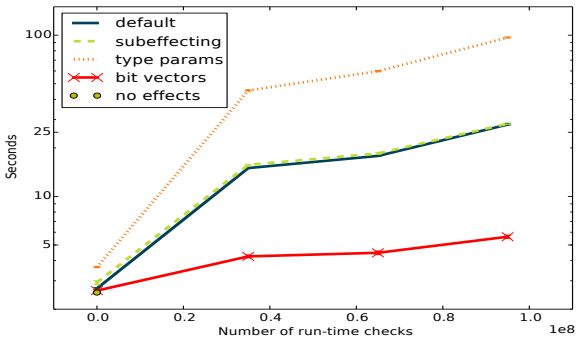

Figure 8. Impact of dynamic checks on execution time (logarithmic scale).

```
app play.* prod @pure
app controllers.* prod @pure
app java.* prod @pure
app org.joda.time.* prod @pure
app scala.(?!Function)* prod @pure
app scala.Function*.tupled* prod @pure
app views.html.helper.* prod @pure
```

Finally, note that the privilege set used to run a program is initially empty, and is adjusted depending on the effect annotations of the `main` method. For example, if `main` is declared as:

```
def main(args: Array[String]): Unit @access
```

then the only available privilege for the execution of the program is `@access`. The default is for the `main` method to grant maximal privileges.

7.3 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 **`restrict`**s 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 8, plotting the total number of runtime checks in each benchmark run with the corresponding time in seconds. Table 1 presents the detailed measurements.

In addition to this baseline experiment (denoted "default" in Figure 8), 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. 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

	# 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 1. Detailed benchmark results

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.

8. Related Work

This work is most related to lightweight effect polymorphism [32] and the theory of gradual effect systems [5], which both build upon the generic effect system of Marino and Millstein [27]. We have already extensively discussed the relation to this work.

There is large body of work of integrating static and dynamic type checking, which we cannot entirely discuss here. The most related work in the area are the efforts to apply the gradual typing approach of Siek and Taha [34] to various advanced typing approaches. These include gradual typestates [16, 40], gradual ownership types [33], gradual security types [12, 15] and gradual typing for annotated type systems [35]. Most of these works are theoretical, while we strive to provide a practical implementation in a widely-used language like Scala.

On the theoretical side, the main challenge of this work is to support effect polymorphic cast. This is related to the work on parametric polymorphism and gradual typing [3, 28], which addresses the issue of casting an unknown function type to a parametrically polymorphic type. In particular, Matthews and Ahmed [28] use runtime sealing to ensure that arguments of a polymorphic type variable are used parametrically. The solution presented in this work is fairly different technically because standard types are properties of values while here we deal with effects, which are properties of computations. But intuitively, one could see the `restrict` operation that is inserted in a polymorphic effect cast as the effect counterpart of the runtime sealing of Matthews and Ahmed.

In spirit, this work is also very close to the motivation of *pluggable type systems* [4, 8, 11]: providing facilities for users to customize the type system as they see fit. In this respect, while existing pluggable type systems are formulated as generic frameworks, our work is specific to effect typing. Most importantly, the customization is extremely easy compared to framework-based approaches: Effscript is a simple declarative language.

Indeed, pluggable type systems can be seen as open implementations [23] of type systems, while the Effscript language is akin to a domain-specific aspect language [14]. In the same way that an aspect language like AspectJ [22] provides a more accessible and user-friendly interface to customize a language than metaobject protocols [21] and open compilers [25], Effscript should be more accessible to non-experts than open type systems.

9. Conclusion

We have presented a generic effect system that is customizable, gradual and polymorphic. The system has been implemented as a compiler plugin for Scala. On the theoretical side, the combination of gradual checking and lightweight effect polymorphism required a subtle redefinition of the rules for higher-order casts, in order to support casts to effect polymorphic functions that preserve the flexibility of effect polymorphism. A full proof of soundness is future work.

This is the first implementation of a gradual generic effect system. Initial performance evaluation show that gradual effect checking can be made practical, even if more optimization opportunities should be exploited.

In addition, another contribution of this work is to address the customizability of effect systems via a simple domain-specific language, Effscript, for declaratively defining effect disciplines. We show the applicability of Effscript with a simple Play application. There are many venues to enhance Effscript, such as augmenting the expressiveness of the pattern matching sub-language, and supporting incremental refinements of existing effect disciplines.

Acknowledgments. We thank Jonathan Aldrich, Felipe Bañados, Jose Daniel Carrasco, Ricardo Jacas, and Gustavo Soto-Ridd for discussions and feedback on either the paper, the artifact, or both. We also thank Jose Ismael Beristain Colorado for his contribution during the initial prototyping of Effscript, as well as the anonymous OOPSLA reviewers for their suggestions to improve the article.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 63–74, San Francisco, CA, USA, Jan. 2008. ACM Press.

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[3] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2011)*, pages 201–214, Austin, Texas, USA, Jan. 2011. ACM Press.

[4] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, pages 57–74, Portland, Oregon, USA, Oct. 2006. ACM Press. ACM SIGPLAN Notices, 41(10).

[5] 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.

[6] N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI 2007)*, pages 15–26. ACM, 2007.

[7] G. M. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In R. Jones, editor, *Proceedings of the 28th European Conference on Object-oriented Programming (ECOOP 2014)*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281, Uppsala, Sweden, July 2014. Springer-Verlag.

[8] G. Bracha. Pluggable type systems. In *Workshop on Revival of Dynamic Languages*, 2004.

[9] 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.

[10] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14, May 2008.

[11] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 681–690, Hawaii, USA, May 2011. ACM Press.

[12] T. Disney and C. Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

[13] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

[14] J. Fabry, T. Dinkelaker, J. Noyé, and É. Tanter. A taxonomy of domain-specific aspect languages. *ACM Computing Surveys*, 47(3):40:1–40:44, Apr. 2015.

[15] L. Fennell and P. Thiemann. Gradual security typing with references. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*, pages 224–239, June 2013.

[16] R. Garcia, É. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12:1–12:44, Oct. 2014.

[17] 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.

[18] C. S. Gordon, W. Dietl, M. D. Ernst, and D. Grossman. Java$_{UI}$: Effects for controlling UI object access. In G. Castagna, editor, *Proceedings of the 27th European Conference on Object-oriented Programming (ECOOP 2013)*, volume 7920 of *Lecture Notes in Computer Science*, pages 179–204, Montpellier, France, July 2013. Springer-Verlag.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2003.

[20] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[21] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[23] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE 97)*, pages 481–490, Boston, Massachusetts, USA, 1997. ACM Press.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[25] J. Lamping, G. Kiczales, L. H. R. Jr., and E. Ruf. An architecture for an open compiler. In *Proceedings of the IMSA 92 Workshop on Reflection and Meta-Level Architectures*, pages 95–106. Akinori Yonezawa and Brian C. Smith, editors, 1992.

[26] D. Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming*. Electronic Proceedings in Theoretical Computer Science, Mar. 2014.

[27] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 39–50, 2009.

[28] J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In S. Drossopoulou, editor, *Proceedings of the 17th European Symposium on Programming Languages and Systems (ESOP 2008)*, volume 4960 of *Lecture Notes in Computer Science*, pages 16–31, Budapest, Hungary, 2008. Springer-Verlag.

[29] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

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

[31] L. Rytz, N. Amin, and M. Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs*, 2013. Article No.: 4.

[32] 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.

[33] I. Sergey and D. Clarke. Gradual ownership types. In H. Seidl, editor, *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 579–599, Tallinn, Estonia, 2012. Springer-Verlag.

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

[35] P. Thiemann and L. Fennell. Gradual typing for annotated type systems. In Z. Shao, editor, *Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP 2014)*, volume 8410 of *Lecture Notes in Computer Science*, pages 47–66, Grenoble, France, 2014. Springer-Verlag.

[36] R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 13–24, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[37] M. Toro and É. Tanter. Gradual polymorphic effects—complete definition and soundness proof. Technical Report TR/DCC-2015-2, University of Chile, July 2015.

[38] Typesafe Inc. Slick functional relational mapping for Scala. `http://slick.typesafe.com/`.

[39] Typesafe, Inc. Play framework, 2007. `http://www.playframework.com/`.

[40] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In M. Mezini, editor, *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483, Lancaster, UK, July 2011. Springer-Verlag.

A. Effect Polymorphic Casts: Detailed Example

Let us consider functions $\texttt{rcv} : ((f\colon \texttt{Int} \xrightarrow{\texttt{@io}} \texttt{Int}) \xrightarrow[f]{\bot} \texttt{Int}) \xrightarrow{\bot} \texttt{Int}$ and function $\texttt{mapl} : (f\colon \texttt{Int} \xrightarrow{¿} \texttt{Int}) \xrightarrow{¿} \texttt{Int}$ defined as follows (for conciseness we omit annotations on lambda terms when these can be easily inferred and/or are not relevant to the argument):

$$\texttt{rcv} = (\lambda g\colon ((f\colon \texttt{Int} \xrightarrow{\texttt{@io}} \texttt{Int}) \xrightarrow[f]{\bot} \texttt{Int}) \,.\, g((\lambda x\colon Int \,.\, x)^{\texttt{Int};\bot;\emptyset}))^{\texttt{Int};\bot;\emptyset}$$

$$\texttt{mapL} = (\lambda f\colon \texttt{Int} \xrightarrow{¿} \texttt{Int} \,.\, f(1))^{\texttt{Int};¿;\emptyset}$$

Suppose an initial privilege set Ξ_i and a program that consists of an application $\texttt{rcv}(\texttt{mapl})$. During transformation of the source language into the internal language, [TApp] inserts casts as follows:

$$\texttt{rcv}(\texttt{mapl}) \Rightarrow \tag{1}$$

$$\langle\!\langle (f\colon \texttt{Int} \xrightarrow{¿} \texttt{Int}) \xrightarrow{¿} \texttt{Int} \xrightarrow{\Xi_i} \texttt{Int} \Leftarrow ((f\colon \texttt{Int} \xrightarrow{\texttt{@io}} \texttt{Int}) \xrightarrow[f]{\bot} \texttt{Int}) \xrightarrow{\bot} \texttt{Int} \rangle\!\rangle^{true}_{\Gamma} \texttt{rcv}(\texttt{mapl}) = \tag{2}$$

$$((\lambda f \,.\, (\lambda x \,.\, \textbf{restrict}\ \Xi_i\ f\bullet(\langle\!\langle (f\colon \texttt{Int} \xrightarrow{\texttt{@io}} \texttt{Int}) \xrightarrow[f]{\bot} \texttt{Int} \Leftarrow (f\colon \texttt{Int} \xrightarrow{¿} \texttt{Int}) \xrightarrow{¿} \texttt{Int} \rangle\!\rangle^{true}_{\Gamma} x))\bullet \texttt{rcv})(\texttt{mapl}) = \tag{3}$$

$$((\lambda f \,.\, (\lambda x \,.\, \textbf{restrict}\ \Xi_i\ f\bullet((\lambda y \,.\, \textbf{restrict}\ \texttt{@io}\ x\bullet(\langle\!\langle \texttt{Int} \xrightarrow{¿} \texttt{Int} \Leftarrow \texttt{Int} \xrightarrow{\texttt{@io}} \texttt{Int} \rangle\!\rangle^{\mathit{false}}_{\Gamma} y)))))\bullet \texttt{rcv})(\texttt{mapl}) = \tag{4}$$

$$((\lambda f \,.\, (\lambda x \,.\, \textbf{restrict}\ \Xi_i\ f\bullet((\lambda y \,.\, \textbf{restrict}\ \texttt{@io}\ x\bullet((\lambda z \,.\, y\bullet_y z)))))\bullet \texttt{rcv})(\texttt{mapl}) \tag{5}$$

Notice how in (4) we restrict the privilege set to $\bot$ plus the latent effects of f. With the naive semantics, we would restrict the privilege set to $\bot$, meaning that any effectful operation would produce a runtime error. Also, because we are casting to an effect-polymorphic function on its argument, we flag the next cast so it does not introduce checks as shown in (5).

Let us consider the case where we insert the **has** as in the naive semantics, as highlighted in (6):

$$((\lambda f \,.\, (\lambda x \,.\, \textbf{restrict}\ \Xi_i\ f\bullet((\lambda y \,.\, \textbf{restrict}\ \texttt{@io}\ x\bullet((\lambda z \,.\, \textbf{has}\ \texttt{@io}\ y\bullet_y z)))))\bullet \texttt{rcv})(\texttt{mapl}) \tag{6}$$

Applying several steps of evaluation we obtain:

$$((\lambda f \,.\, (\lambda x \,.\, \textbf{restrict}\ \Xi_i\ f\bullet((\lambda y \,.\, \textbf{restrict}\ \texttt{@io}\ x\bullet((\lambda z \,.\, \textbf{has}\ \texttt{@io}\ y\bullet_y z)))))\bullet_{\Gamma} \texttt{rcv})(\texttt{mapl}) \rightarrow \tag{7}$$

$$(\lambda x \,.\, \textbf{restrict}\ \Xi_i\ \texttt{rcv}\bullet((\lambda y \,.\, \textbf{restrict}\ \texttt{@io}\ x\bullet((\lambda z \,.\, \textbf{has}\ \texttt{@io}\ y\bullet_y z)))))(\texttt{mapl}) \rightarrow \tag{8}$$

$$\textbf{restrict}\ \Xi_i\ \texttt{rcv}\bullet((\lambda y \,.\, \textbf{restrict}\ \texttt{@io}\ \texttt{mapl}\bullet((\lambda z \,.\, \textbf{has}\ \texttt{@io}\ y\bullet_y z)))) \rightarrow \tag{9}$$

$$\textbf{restrict}\ \Xi_i\ (\lambda y \,.\, \textbf{restrict}\ \texttt{@io}\ \texttt{mapl}\bullet((\lambda z \,.\, \textbf{has}\ \texttt{@io}\ y\bullet_y z)))(\lambda x \,.\, x) \rightarrow \tag{10}$$

$$\textbf{restrict}\ \Xi_i\ \textbf{restrict}\ \texttt{@io}\ \texttt{mapl}\bullet((\lambda z \,.\, \textbf{has}\ \texttt{@io}\ (\lambda x \,.\, x)\bullet_y z)) \rightarrow \tag{11}$$

$$\textbf{restrict}\ \Xi_i\ \textbf{restrict}\ \texttt{@io}\ (\lambda z \,.\, \textbf{has}\ \texttt{@io}\ (\lambda x \,.\, x)\bullet_y z)(1) \rightarrow \tag{12}$$

$$\textbf{restrict}\ \Xi_i\ \textbf{restrict}\ \texttt{@io}\ \textbf{has}\ \texttt{@io}\ (\lambda x \,.\, x)\bullet_y 1 \tag{13}$$

If the initial privilege set Ξ_1 is $\bot$ then the evaluation leads to `Error` even though no effect is produced. As explained before, when the target of a cast is an effect polymorphic function on its argument, the insertion of **has** is not needed as it only represents the maximal privilege set that the polymorphic function accepts as argument, not what the actually-produced effect is. In this example the appropriate check for permissions is done while type checking the body of $\texttt{rcv}\colon g((\lambda x\colon Int \,.\, x))$

B. Effscript Syntax

The syntax of Effscript is presented in Figure 9.

effscript	::=	*name effects lattice* [*effspecs*]	Effscript
name	::=	**name:** ⟨*string*⟩	Effscript name
effects	::=	**effects:** *effectDef*+	Effscript effects
lattice	::=	**lattice:** *top bottom relation*∗	Effscript lattice
effspecs	::=	**effspecs:** *effspec*+	Effscript effspecs
top	::=	**top:** *effect*+	Top effect
bottom	::=	**bottom:** *effect*	Bottom effect
relation	::=	*effect* **<:** *effect*	Effect relation
effectDef	::=	**@unknown** | **@** ⟨*string*⟩ [*typeParamDef*]	Effect privilege definition
effect	::=	**@unknown** | **@** ⟨*string*⟩ [*typeParamComp*]	Effect privilege
typeParamComp	::=	**[** *typeCompLess*+$^{(,)}$**]**	Type param with comparison op.
typeParamDef	::=	**[** ([*variance*] *typeCompLess*)+$^{(,)}$**]**	Type param with comparison op. and variance
typeParam	::=	**[** *typeVariable*+$^{(,)}$**]**	Type param
variance	::=	**+** | **-**	Type variance
typeCompLess	::=	*typeVariable* [**<:** *typeName*]	Type with comparison op.
typeComp	::=	*typeVariable* [*comp typeName*]	Type with comparison op.
comp	::=	**<:** | **:>** | **=:=**	Comparison operator
effspec	::=	*pointcut advice*+	External effect specification
pointcut	::=	*selector signature*	Pointcut
selector	::=	**app** | **def**	Selector
signature	::=	*pathPattern typeParam* [*args*]	Function signature
args	::=	**(** *typeComp*+$^{(,)}$**)**	Function argument types
advice	::=	*operation effect* [*within*]	Effect operation
operation	::=	**prod** | **ann**	Operation
within	::=	**within** *pathPattern*	Lexical within
pathPattern	::=	pattern that denotes a path, such as `scala.collection`$\star$, $\star$`.println`	Path pattern (*)
typeVariable	::=	any type variable, such as `T` or `V`	Type variable
typeName	::=	name that denotes a type, such as `String` or `Int`	Type name

(*) : The $\star$ is used as a wildcard to represent any sequence of characters excluding the `.`.

Figure 9. Effscript syntax