Lightweight Polymorphic Effects

Lukas Rytz, Martin Odersky, and Philipp Haller

EPFL, Switzerland,
{first.last}@epfl.ch

Abstract. Type-and-effect systems are a well-studied approach for reasoning about the computational behavior of programs. Nevertheless, there is only one example of an effect system that has been adopted in a wide-spread industrial language: Java's checked exceptions. We believe that the main obstacle to using effect systems in day-to-day programming is their verbosity, especially when writing functions that are polymorphic in the effect of their argument. To overcome this issue, we propose a new syntactically lightweight technique for writing effect-polymorphic functions. We show its independence from a specific kind of side-effect by embedding it into a generic and extensible framework for checking effects of multiple domains. Finally, we verify the expressiveness and practicality of the system by implementing it for the Scala programming language.

1 Introduction

Type-and-effect systems are a well understood and widely used approach in the research literature for reasoning about computational effects. Originally designed to delimit the scope of dynamically allocated memory [21], the technique has been applied to various kinds of effects such as exceptions [8], purity [18], atomicity [1] or parallel programming [2]. Marino et al. [14] factor out the commonalities of different effect systems into a generic framework.

However, when taking a look at the most wide-spread programming languages used in industry, there is only one example of an effect system that has been put into practice: Java's checked exceptions. In addition, this particular system has earned a lot of critique about its verbosity and lack of expressiveness ([10], [22]), which in turn influenced language designers not to put effect systems into their languages ([10], [16]).

We believe that the fundamental property that makes effect systems expressive enough to be useful in everyday programming is effect-polymorphism. Functions are often implemented using delegation, by calling functions they receive as argument, and therefore the effect of a function can depend on the effect its arguments. Polymorphic effect systems have been around for more than 20 years [13], and within limits,[1] Java also supports methods that are polymorphic in the thrown exception type.

[1] It is only possible to abstract over a fixed number of exception types

```
abstract class List<T> {
 public <U> List<U> mapM(Function<T, U> f) throws Exception
 public <U, E extends Exception> List<U> mapP(FunctionE<T, U, E> f) throws E
}
```

This example shows two higher-order methods `mapM` and `mapP` in Java. The monomorphic version `mapM` can only accept functions with arbitrary effects as argument if it declares the effect `throws Exception`. The second version is polymorphic in the exception type of its argument function. Note that also the function type `FunctionE` has to be extended with an explicit exception type parameter.

The crucial issue is that writing effect-polymorphic methods results in code which is syntactically heavy and hard to understand. Also, polymorphism is tied to one specific effect domain: adding a new kind of effect system to Java would often also require to integrate a new syntax for effect-polymorphism.

In this paper, we propose a pragmatic and expressive new way for writing effect-polymorphic code. We present a system for lightweight polymorphic effect checking with the following contributions:

– We propose a syntax for writing effect-polymorphic functions which is as lightweight as writing an ordinary, monomorphic function, without the need for explicit effect parameters.
– To support these functions in the type system, we introduce a new kind of function type for effect-polymorphic functions. These types co-exist with ordinary function types. In a monomorphic function type $T \overset{e}{\Rightarrow} U$, the latent effect, the effect that might occur when the function is invoked, is annotated as e. For an effect-polymorphic function type $T \overset{e}{\rightarrow} U$, the latent effect consists of both e and the latent effect of its argument type T.
– We embed the new kind of functions into a generic effect checking framework which is independent of a specific effect domain. This extends the generic effect system proposed in [14] with effect-polymorphism. The framework is extensible and allows checking multiple kinds of effects at the same time, given a description of each effect domain. We show that every effect domain profits from the effect-polymorphism available through the framework.
– The described framework is implemented as a compiler plugin for the Scala programming language. We added effect checking for exceptions and successfully applied polymorphic effects to the core of the Scala collections library.

The rest of this paper is structured as follows. Section 2 gives an informal overview of the system, Section 3 presents the formalization, in Section 4 we report our practical experience, Section 5 discusses related work and Section 6 concludes.

2 Overview

In this section, we give an informal overview of our polymorphic type-and-effect system and we show that it constitutes a pragmatic and practical compromise between syntactic simplicity and expressivity.

2.1 Effect-polymorphic Function Types

The main idea of our type-and-effect system is to introduce a new kind of function type which, by definition, denotes effect-polymorphic functions. A function is said to be effect-polymorphic if its *latent* effect, the effect that occurs when the function is applied, depends on the effect of its argument. To give an example, we define a simple higher-order function `hof` which applies its argument function to the constant `1`.

```
val hof = (f: Int ⇒ Int) → f 1
```

Intuitively, the effect of applying `hof` to a function `f` depends on the effect of the argument `f`. For instance, if "`f = (x: Int) ⇒ x + 1`" is a pure function,[2] then the invocation of `hof` does not have any side-effect. But if we apply it to a function "`f = (x: Int) ⇒ throw ex`" that throws an exception, then so does the invocation. We therefore conclude that the function `hof` is effect-polymorphic in its argument function `f`.

To differentiate between effect-polymorphic and ordinary, effect-monomorphic functions, we use two kinds of arrows in function types. The double arrow $\Rightarrow$ is used for ordinary function types. For instance, the function `(x: Int) ⇒ x + 1` has type $\texttt{Int} \overset{\perp}{\Rightarrow} \texttt{Int}$. The effect annotation on the arrow denotes the latent effect of the function, the effect that may occur when the function is applied. The symbol $\perp$ denotes purity. If the effect annotation is omitted, the largest possible effect $\top$ is assumed.

An effect-polymorphic function type, such as the type of `hof`, is expressed with a single arrow $\rightarrow$. Like ordinary function types, also polymorphic function types are annotated with an effect, however the default effect when the annotation is omitted is $\perp$. So the function `hof` has the following type:

$$\texttt{hof: (Int} \overset{\top}{\Rightarrow} \texttt{Int)} \overset{\perp}{\rightarrow} \texttt{Int // } \textit{equivalent to (Int} \Rightarrow \textit{Int)} \rightarrow \textit{Int}$$

The crucial property of an effect-polymorphic function type is that its latent effect consist of two components:

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

The second component is the source of effect-polymorphism. For each invocation of the polymorphic function, the effect of the argument can be different, which results in a different overall effect. We take a closer look at the types of the two function literals from the previous example:

$$\texttt{val f: (Int} \overset{\perp}{\Rightarrow} \texttt{Int) = (x: Int)} \Rightarrow \texttt{x + 1}$$
$$\texttt{val g: (Int} \xRightarrow{\texttt{throw(ex)}} \texttt{Int) = (x: Int)} \Rightarrow \texttt{throw ex}$$

[2] Instead of using the traditional abstraction syntax $\lambda x : T.t$, we write function literals in the form $(x : T) \Rightarrow t$

For each invocation of the function `hof`, the effect gets computed based on the actual argument type. Therefore, the invocation "`hof f`" has no effect, while the invocation "`hof g`" has the effect of throwing an exception.

2.2 Programming with Effect-polymorphic Functions

We believe that the introduction of effect-polymorphic function types is a very efficient technique for adding polymorphic effect checking to a programming language, while keeping the annotation overhead for programmers within reasonable limits. To illustrate this point, we look at the higher-order function `map` which applies a given function to all elements of a list.

```
val map: IntList $\overset{\perp}{\Rightarrow}$ (Int ⇒ Int) → IntList =
  (l: IntList) ⇒ (f: Int ⇒ Int) → l match {
    case Nil       => Nil
    case Cons x xs => Cons (f x) (map xs f)
  }
```

Even though the signature of the `map` function is fully effect-polymorphic, there is only one single effect annotation $\perp$, which denotes purity of the outer function. Since pure functions are very common, especially when currying is used to encode functions with multiple arguments, it might be worthwhile to introduce syntactic sugar for pure functions. In this case, the polymorphic `map` function would not need any effect annotation at all.

There are many higher-order functions that can be made effect-polymorphic by replacing a normal function type with an effect-polymorphic one. However, the type system we introduce does not only apply to functional programming. On the contrary: the presented ideas are based on our work on a polymorphic effect system for the Scala programming language. We decided to present the system using a simpler lambda calculus in order not to distract from the main concepts. In Section 4 we show how lightweight effect-polymorphism can be expressed in the object-oriented setting, and we present the results obtained with our implementation for Scala in Section 4.3.

Effect-polymorphism is a crucial ingredient to make effect-checking practicable in a real-world programming language, and this applies equally to object-oriented languages. For instance, many of the object-oriented design patterns identified in [5] are based on delegation. In order to correctly annotate the effect of methods which are implemented by calling methods of their parameters, polymorphic effect annotations are indispensable. The widely used strategy pattern is the most prominent example: it basically models higher-order functions. A method that is implemented in terms of its argument strategy is polymorphic in the effect of that strategy.

The effect system for checked exceptions in Java illustrates the need for effect-polymorphism. It is possible in Java to write methods that are polymorphic in the thrown exception type, but doing so is very verbose and therefore often avoided in practice. This limitation is at the source of the well-known issues with Java's checked exceptions ([15], [22]): `throws` declarations are often copy-pasted from the callee to the caller method, which has a negative impact on the maintainability and readability of the program code.

2.3 An Extensible Framework for Multiple Effect Domains

The polymorphic type-and-effect system outlined in the previous section is not tied to a specific kind of side effect, such as exceptions that might be thrown or state that might be modified. Instead, it defines an extensible framework that allows effect checking of multiple effect domains in the same language, at the same time.

In order to add a new kind of side effect to be checked by the framework, a description of the effect domain in the form of a semi-lattice has to be provided. The semi-lattice consists of a set of effects for the domain, a join operation to compute the combination of two effects, and a sub-effect relation which compares two effects.[3] The following example shows a simple effect lattice for tracking IO effects:

- Effect set: $E_{\mathcal{I}} = \{noIO, IO\}$
- Join operation: $e_1 \sqcup_{\mathcal{I}} e_2 = \begin{cases} IO & \text{if } (e_1 = IO) \vee (e_2 = IO) \\ noIO & \text{otherwise} \end{cases}$
- Sub-effect relation: $e_1 \sqsubseteq_{\mathcal{I}} e_2 \;=\; (e_1 = noIO) \vee (e_2 = IO)$

The framework also needs to know the top and bottom elements of each effect lattice. In the case of IO effects, these are $\top_{\mathcal{I}} = IO$ and $\bot_{\mathcal{I}} = noIO$.

In addition to the effect lattice, every concrete effect domain has to define the effect associated with each syntactic construct of the language. For instance, an effect system for tracking exceptions declares that a `throw` expression adds an effect, and that a `try` expression can mask effects. This information has to be provided in the form of a function $\mathit{eff}_{\mathcal{D}}$ which receives as argument a representation of the program fragment in question and returns its side-effect.[4] This function is closely related to the "adjust" function in [14], and we will give a precise definition in Section 3.3.

In the domain of IO, effects are introduced by calling pre-defined functions that have a latent IO effect. There are no syntactic constructs that introduce or mask IO effects, therefore the $\mathit{eff}_{\mathcal{I}}$ function can be left unspecified and the framework will use a default definition.

Annotating Multiple Effect Domains Since the effect checking framework supports tracking effects from multiple domains, the effect annotations on function types have to declare an effect for every domain that is being checked. This

[3] Note that the lattice operations need to fulfill the common lattice properties, such as transitivity for $\sqsubseteq$. Also, the join and sub-effect operations are related: for all effects e_1, e_2, we have $e_1 \sqsubseteq e_2 \iff e_1 \sqcup e_2 = e_2$.

[4] We will discuss the effect domain of exceptions in detail in Section 3.4

is achieved by annotating the function types with a tuple consisting of domain-specific effect annotations. For instance, if there are three effect domains $\mathcal{D}1$, $\mathcal{D}2$ and $\mathcal{D}3$, a function type has the form $T_1 \xRightarrow{e_{\mathcal{D}1}\ e_{\mathcal{D}2}\ e_{\mathcal{D}3}} T_2$.

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

- The function `map` accepts as argument a function that can have any effect. Therefore, this argument is annotated with the topmost effect.
- The implementation of `map` itself is pure, there is no other effect than the effect of its argument.
- The `map` function is only one example of a large class of library functions that are pure. The same is true for instance for all operations on immutable datatypes.
- Functions that do have side-effects usually have effects in one or very few effect-domains. For instance, a random generator is non-deterministic, operations on mutable data structures modify state, or functions from a file-API have IO-effects. In addition, these functions might have exceptional behavior. However, it is rather uncommon to have functions with side-effects from all those domains at the same time.

In order to simplify the multi-domain effect annotations and take the above observations into account, we introduce two specific effect annotations which, by definition, range over all effect domains: $\top$ and $\bot$. When used as such, the two annotations have the expected meaning: $\top = \top_{\mathcal{D}1}\ \ldots\ \top_{\mathcal{D}N}$, similarly for $\bot$.

The crucial characteristic however is that the multi-domain annotations can be combined with concrete effect annotations from individual effect domains. For instance, the type $T_1 \xRightarrow{\bot\ e_{\mathcal{D}i}} T_2$ denotes a function which can have effect $e_{\mathcal{D}i}$ in the domain $\mathcal{D}i$, but is pure in all other domains. Similarly, combining the $\top$

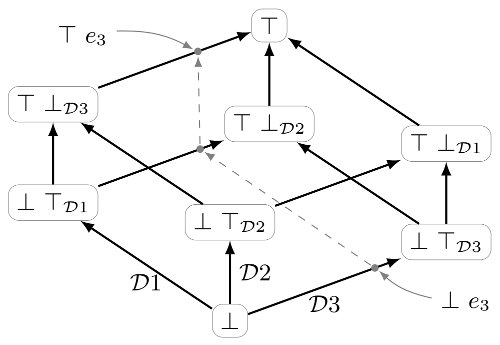
$\top\ e_3$
$\top$
$\top\ \bot_{\mathcal{D}3}$
$\top\ \bot_{\mathcal{D}2}$
$\top\ \bot_{\mathcal{D}1}$
$\bot\ \top_{\mathcal{D}1}$
$\bot\ \top_{\mathcal{D}2}$
$\bot\ \top_{\mathcal{D}3}$
$\mathcal{D}1$
$\mathcal{D}2$
$\mathcal{D}3$
$\bot\ e_3$
$\bot$

Fig. 1. Effect annotations in multiple domains

annotation with concrete effects restricts the allowed effect in certain domains. This behavior is illustrated in Figure 1, showing an example with three effect domains.

3 Formalization

In order to formalize the ideas presented in the previous section, we extend a simply typed lambda calculus with effect annotations and effect-polymorphic function types. The syntax of the formal language is summarized in Figure 2. Note that the syntax for function abstraction is different than usual and does not use the λ symbol.

t	$::= x$	parameter
	$\mid t\ t$	application
	$\mid v$	value
v	$::= (x : T) \Rightarrow t$	monomorphic abstraction
	$\mid (x : T) \rightarrow t$	effect-polymorphic abstraction
T	$::= T \overset{e}{\Rightarrow} T$	function type
	$\mid T \overset{e}{\rightarrow} T$	effect-polymorphic function type
e	$::= \bot\ e_D \mid \top\ e_D \mid e_D$	effect annotation
e_D	$::= e_D\ e_D \mid \cdot$	concrete effects
Γ	$::= \emptyset \mid \Gamma, x : T$	parameter context
f	$::= \epsilon \mid x$	polymorphism context

Fig. 2. Core language syntax

The effect annotation e on a function type declares the *latent* effect, the effect that may occur when the function is invoked. Note that e defines an effect for every active effect domain. Subsection 3.1 explains how the integration of multiple effect domains into one effect system is handled.

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

A monomorphic function type $T_1 \overset{e}{\Rightarrow} T_2$ declares a latent effect e, the effect that may occur when the function is applied. In the type of an effect-polymorphic function $T_1 \overset{e}{\rightarrow} T_2$ however, the annotated effect e does not denote the entire latent effect of the function. Instead, the effect of such a function consists of two parts: the concrete, annotated effect e and the effect of its argument of type T_1. Only higher-order functions, functions that take another function as argument, can be effect-polymorphic. This invariant is checked by the typing rule T-ABS-POLY which enforces the parameter type T_1 to be a function type.

The effect of the argument function is implicitly added to the total effect of an effect-polymorphic function.

For syntactic convenience, the effect annotations on function types can be omitted, in which case the following default effects are used:

- $T_1 \Rightarrow T_2$ is a equivalent to $T_1 \overset{\top}{\Rightarrow} T_2$
- $T_1 \rightarrow T_2$ is a equivalent to $T_1 \xrightarrow{\bot} T_2$

Example 1. We inspect the type of the simple higher-order function `hof` introduced in Section 1:

$$\texttt{val hof: (Int} \overset{\top}{\Rightarrow} \texttt{Int)} \xrightarrow{\bot} \texttt{Int = (f: Int} \overset{\top}{\Rightarrow} \texttt{Int)} \rightarrow \texttt{f 1}$$

Using the default effects mentioned above, the effect annotations in the type of `hof` as well as the one in the function abstraction can be omitted:

```
val hof: (Int ⇒ Int) → Int = (f: Int ⇒ Int) → f 1
```

3.1 A Multi-domain Effect Lattice

When checking multiple kinds of effects at the same time, every effect domain needs to be described as a join-semilattice as explained in Section 2.3. For a domain $\mathcal{D}$, the lattice consists of a set of atomic effects $E_\mathcal{D}$, a join operation $\sqcup_\mathcal{D}$ and a sub-effect relation $\sqsubseteq_\mathcal{D}$. Additionally, the bottom and top elements $\bot_\mathcal{D}$ and $\top_\mathcal{D}$ of $E_\mathcal{D}$ have to be specified.

These individual domains are combined into one multi-domain effect lattice that is used in this section. The elements of this lattice are tuples of effects from the individual domains:

$$E = \{e_{\mathcal{D}1} \dots e_{\mathcal{D}n} \mid e_{\mathcal{D}1} \in E_{\mathcal{D}1} \wedge \dots \wedge e_{\mathcal{D}n} \in E_{\mathcal{D}n}\}$$

The $\sqcup$ and $\sqsubseteq$ operations are defined element-wise using $\sqcup_{\mathcal{D}i}$ and $\sqsubseteq_{\mathcal{D}i}$ for every domain $\mathcal{D}i$. We omit their definitions here for brevity.

3.2 Subtyping

The subtyping relation of our calculus has the common reflexivity and transitivity properties.

$$\frac{}{T <: T}\ (\textsc{S-Refl}) \qquad \frac{T' <: S \qquad S <: T}{T' <: T}\ (\textsc{S-Trans})$$

The subtyping rules covering the two kinds of function types in our system are entirely symmetrical.

$$\frac{T_1 <: T_1' \qquad T_2' <: T_2 \qquad e' \sqsubseteq e}{T_1' \overset{e'}{\Rightarrow} T_2' <: T_1 \overset{e}{\Rightarrow} T_2}\ (\textsc{S-Fun-Mono}) \qquad \frac{T_1 <: T_1' \qquad T_2' <: T_2 \qquad e' \sqsubseteq e}{T_1' \xrightarrow{e'} T_2' <: T_1 \xrightarrow{e} T_2}\ (\textsc{S-Fun-Poly})$$

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

```
val pureHof = (f: Int ⇒ Int) ⇒ f 1
```

$\texttt{val pureHof = (f: Int} \overset{\bot}{\Rightarrow} \texttt{Int)} \Rightarrow \texttt{f 1}$

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

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

$\texttt{val maybePure: (Int} \overset{\top}{\Rightarrow} \texttt{Int)} \xrightarrow{\bot} \texttt{Int = ...}$

$\texttt{val pure: (Int} \overset{\bot}{\Rightarrow} \texttt{Int)} \xrightarrow{\bot} \texttt{Int = ...}$

In general, an invocation of `maybePure` might have any effect, while an invocation of `pure` is always pure. However, the subtyping relation seems to contradict this observation: due to contra-variance of arguments, the type of `maybePure` is a subtype of the type of `pure`.

To build an intuition why the subtyping rule is correct, we take a closer look at the two function types. The type of `pure` says: "Give me a pure function from `Int` to `Int`, and I compute a result without producing a side-effect." For instance, in the body of a function `m`

$\texttt{val m = (pure: (Int} \overset{\bot}{\Rightarrow} \texttt{Int)} \xrightarrow{\bot} \texttt{Int)} \Rightarrow \texttt{...}$

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

```
m maybePure
```

As explained before, this is allowed by the subtyping rules. We can now see that it is also correct, because in the body of method `m` only pure functions will be passed into `maybePure`. Due to effect-polymorphism, those invocations of `maybePure` have no effect. In other words, the type of the function `maybePure` says: "If you give me a pure function, I *also* compute a result without producing a side-effect!"

3.3 Static Semantics

Extensible Type-and-effect Checking Since we are creating an extensible framework for tracking side-effects of multiple effect domains, we want to give each concrete effect system the possibility of customizing the effect of evaluating a term. For that reason, the typing rules introduced in this section are parametrized by an auxiliary function *eff*.

For every effect domain $\mathcal{D}$, the function $\mathit{eff}_{\mathcal{D}}$ computes the effect of evaluating a term, given the effects of its sub-terms. It takes two arguments: a name indicating the syntactic form in question, and a list of effects of its sub-terms. By default it combines all the argument effects using the $\sqcup_{\mathcal{D}}$ operator:

$$\mathit{eff}_{\mathcal{D}}(*, \overline{e}) = \textstyle\bigsqcup_{\mathcal{D}} \overline{e}$$

The default $\mathit{eff}_{\mathcal{D}}$ function can be specialized by concrete effect domains; we will discuss the example of exceptions in Section 3.4. However, arbitrary $\mathit{eff}_{\mathcal{D}}$ functions can make the type-and-effect system unsound. For the system to be correct, the $\mathit{eff}_{\mathcal{D}}$ functions needs to meet the following monotonicity requirement.

Lemma 1. *Monotonicity.*
For every effect domain $\mathcal{D}$ and every syntactic form Trm*,*

1. *if* $\forall\ e_i \in e, d_i \in d\ .\ e_i \sqsubseteq d_i$*, then* $\mathrm{eff}_{\mathcal{D}}(\textsc{Trm}, \overline{e}) \sqsubseteq \mathrm{eff}_{\mathcal{D}}(\textsc{Trm}, \overline{d})$
2. $\mathrm{eff}_{\mathcal{D}}(\textsc{Trm}, e_1, \ldots, e_{i1} \sqcup e_{i2}, \ldots, e_n) \sqsubseteq \mathrm{eff}_{\mathcal{D}}(\textsc{Trm}, e_1, \ldots, e_{i1}, \ldots, e_n) \sqcup e_{i2}$

The first clause of the monotonicity lemma requires the $\mathit{eff}_{\mathcal{D}}$ functions to be monotonic. Implementing effect masking remains possible, as we will see in the effect domain of exceptions. The second part of the monotonicity lemma prevents the output effect to depend on the presence of a certain input effect. This restriction falls in line with the general semantics of effect annotations in type-and-effect systems: an annotated effect *may* occur, but it is not required to occur.

The function eff used in the typing statements works on all effect domains at the same time, similar to the multi-domain lattice operations described in Section 3.1. It is composed of the individual $\mathit{eff}_{\mathcal{D}i}$ functions in the straightforward way:

$$\mathit{eff}(\textsc{Trm}, \overline{e}) = \mathit{eff}_{\mathcal{D}1}(\textsc{Trm}, \overline{e})\ \ldots\ \mathit{eff}_{\mathcal{D}n}(\textsc{Trm}, \overline{e})$$

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

As is common, referencing a parameter does not have a side-effect:

$$\frac{x : T \in \Gamma}{\Gamma; f\ \vdash\ x : T\ !\ \bot} \qquad \text{(T-Param)}$$

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

$$\frac{\Gamma, x : T_1; \epsilon\ \vdash\ t : T_2\ !\ e}{\Gamma; f\ \vdash\ (x : T_1) \Rightarrow t : T_1 \overset{e}{\Rightarrow} T_2\ !\ \bot} \ \text{(T-Abs-Mono)} \qquad \frac{\begin{array}{c}\Gamma; f\ \vdash\ t_1 : T_1 \overset{e}{\Rightarrow} T\ !\ e_1 \\ \Gamma; f\ \vdash\ t_2 : T_2\ !\ e_2 \qquad T_2 <: T_1\end{array}}{\Gamma; f\ \vdash\ t_1\ t_2 : T\ !\ \mathit{eff}(\textsc{App}, e_1, e_2, e)} \ \text{(T-App-Mono)}$$

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

The rule T-App-Mono is a standard typing rule for method applications. The resulting effect consists of three parts: e_1 is the effect of evaluating the function, e_2 is the effect of evaluating the argument and e is the latent effect of the function. These three effects are combined using the *eff* function introduced in the beginning of this section.[5]

Next we analyze the typing rules for effect-polymorphic function abstractions and invocations. But before that, we take a close look at the functionality of the extended typing environment $\Gamma; f$.

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

```
val hof = (f: Int ⇒^⊤ Int) → f 1
```

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

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

$$\frac{T_1 = T_a \overset{e_1}{\Rightarrow} T_b \qquad \Gamma, f : T_1; f \vdash t : T_2 \ ! \ e}{\Gamma; f' \vdash (f : T_1) \rightarrow t : T_1 \xrightarrow{e} T_2 \ ! \ \bot} \quad \text{(T-Abs-Poly)}$$

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

The following typing rule T-App-Param implements the mentioned special treatment of the argument function f:

$$\frac{\begin{array}{c} f : T_1 \overset{e}{\Rightarrow} T \in \Gamma \\ \Gamma; f \vdash t : T_2 \ ! \ e_2 \qquad T_2 <: T_1 \end{array}}{\Gamma; f \vdash f \ t : T \ ! \ \mathit{eff}(\textsc{App}, \bot, e_2, \bot)} \quad \text{(T-App-Param)}$$

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

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

$$\frac{\begin{array}{c} \Gamma; f \vdash t_1 : T_1 \xrightarrow{e} T \ ! \ e_1 \\ \Gamma; f \vdash t_2 : T_2 \ ! \ e_2 \qquad T_2 <: T_1 \end{array}}{\Gamma; f \vdash t_1 \ t_2 : T \ ! \ \mathit{eff}(\textsc{App}, e_1, e_2, e \sqcup \mathit{latent}(T_2))} \quad \text{(T-App-Poly)}$$

There is one single but crucial difference to the rule T-App-Mono for monomorphic function applications. The latent effect of the function t_1 consists of two

[5] Remember that by default, *eff* computes the join of its argument effects

components: the concrete effect e annotated in the function type, and the latent effect of the argument function t_2 which is computed using $\mathit{latent}(T_2)$.

Note that the rule T-App-Poly is at the root of our effect-polymorphic type system. We obtain effect-polymorphism by computing for each invocation of t_1 the effect of the actual argument type T_2.

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

$$\mathit{latent}(T) = e \qquad \text{where } T = T_1 \overset{e}{\Rightarrow} T_2$$

The reason why only monomorphic functions are allowed as parameters of effect-polymorphic functions is that the typing rules become simpler without decreasing the expressiveness. Imagine that a polymorphic function takes another polymorphic function as argument:

```
val highHof = (f: (Int ⇒ Int) → Int) → f ((x: Int) ⇒ x + 1)
```

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

3.4 Examples of Concrete Effect Domains

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

Exceptions In order to add effect checking for exceptions, we first need to extend the base language with primitives to throw and handle exceptions. The additions to the language syntax are presented in Figure 3. We use a finite set of exceptions $p_1 \ldots p_n$ that can be thrown and caught, however the system could be easily extended to an open hierarchy of effects such as the exception types in languages like Scala or Java. An effect annotation $\texttt{throws}(\overline{p})$ denotes that any of the exceptions in $\overline{p}$ might be thrown. The effect lattice for the exception domain $\mathcal{E}$ is defined in Figure 4.

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

$$\frac{}{\texttt{Nothing} <: T} \qquad \text{(S-Nothing)}$$

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

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

Fig. 3. Extended syntax for exceptions

$$E_{\mathcal{E}} = \{\mathtt{throws}(p) \mid p \subseteq \{p_1, \ldots, p_i\}\} \qquad \mathtt{throws}(\overline{p}) \sqsubseteq_{\mathcal{E}} \mathtt{throws}(\overline{q}) \iff p \subseteq q$$
$$\bot_{\mathcal{E}} = \mathtt{throws}() \qquad \mathtt{throws}(\overline{p}) \sqcup_{\mathcal{E}} \mathtt{throws}(\overline{q}) = \mathtt{throws}(\overline{p} \cup \overline{q})$$
$$\top_{\mathcal{E}} = \mathtt{throws}(p_i, ..., p_n)$$

Fig. 4. Effect lattice for exceptions

$$\frac{e = \mathit{eff}(\textsc{Throw}(p))}{\Gamma; f \vdash \mathtt{throw}(p) : \mathtt{Nothing}\ !\ e} \qquad (\text{T-}\textsc{Throw})$$

$$\frac{\begin{array}{cc} \Gamma; f \vdash t_1 : T_1\ !\ e_1 & T_1 <: T \\ \Gamma; f \vdash t_2 : T_2\ !\ e_2 & T_2 <: T \\ e_t = \mathit{eff}(\textsc{Try}, e_1) & e = \mathit{eff}(\textsc{Catch}(\overline{p}), e_t, e_2) \end{array}}{\Gamma; f \vdash \mathtt{try}\ t_1\ \mathtt{catch}(\overline{p})\ t_2 : T\ !\ e} \qquad (\text{T-}\textsc{Try})$$

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

$$\begin{array}{ll} \mathit{eff}_{\mathcal{E}}(\textsc{Throw}(p)) & = \mathtt{throws}(p) \\ \mathit{eff}_{\mathcal{E}}(\textsc{Try}, e) & = e \\ \mathit{eff}_{\mathcal{E}}(\textsc{Catch}(\overline{p}), e_1, e_2) & = \mathtt{throws}((\overline{q} \setminus \overline{p}) \cup \overline{s}) \quad \text{where } \mathtt{throws}(\overline{q}) \in e_1 \\ & \qquad\qquad\qquad\qquad\qquad\quad \mathtt{throws}(\overline{s}) \in e_2 \end{array}$$

Asynchronous Operations The second extension that we present can be used to check the correct and/or efficient use of asynchronous computations. Several popular languages, including C#, F# [20], and Scala, support constructs to start a computation asynchronously, returning a handle (typically called a "future") used to retrieve the result, once it becomes available.

For example, in Scala a long-running computation can be started as follows:

```
val ft = future {
  // long-running computation
}
```

For efficiency, the body of `future` is executed on a thread pool. `ft` is a handle for the result; when retrieving its result, it blocks the current thread until the

t	$::=$ ...	
	$\mid$ `future` t	asynchronous expression
	$\mid$ `block`	blocking expression
	$\mid$ `blocking` t	delimiting blocking expression
e_D	$::=$ ...	
	$\mid$ `B` $\mid$ `noB`	blocking / non-blocking effect annotation

Fig. 5. Extended syntax for asynchronous operations

future's result has been computed or an unhandled exception has been thrown in the future's body:

```
val result = ft()
```

Retrieving the `result` as shown above is a blocking operation. However, when run using a fixed-size thread pool, calling blocking operations inside the bodies of futures is problematic. Blocking operations are operations that may cause the underlying thread to wait indefinitely. Examples are waiting for the completion of a synchronous I/O operation, or waiting on a condition variable inside a monitor. Such thread-blocking operations may lead to starvation, and, in the worst case, may lock up the entire thread pool [9].

Using an effect system, it is possible to prevent these system-induced deadlocks at compile time. The idea is to require wrapping blocking operations, so that the underlying thread pool can be resized temporarily. This approach to supporting blocking operations has been adopted in the fork/join pool of Java 7 [12]. In the following we present an effect system which guarantees that all blocking operations are properly wrapped, thereby eliminating an entire class of concurrency errors when using thread pools.

The additions to the language syntax are presented in Figure 5. For simplicity, we use a fixed blocking expression `block`; in practice, many more expressions could be potentially blocking, for example, functions for synchronization or blocking I/O. The `future` t expression asynchronously runs an expression t which must be non-blocking, i.e., pure in this effect system. The fact that an expression is blocking is expressed using the `B` effect annotation. We omit the definition of the effect lattice, since it is trivial in this case. Finally, the `blocking` t expression wraps a potentially blocking expression t, such that the effect of the wrapped expression is non-blocking.

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

$$\frac{\Gamma; \epsilon \vdash t : T \,!\, e \qquad \mathtt{B} \notin e}{\Gamma; f \vdash \mathtt{future}\ t : T \,!\, \mathit{eff}(\textsc{Future}, e)} \qquad (\text{T-}\textsc{Future})$$

$$\frac{}{\Gamma; f \vdash \mathtt{block} : T \,!\, \mathit{eff}(\textsc{Block})} \qquad (\text{T-}\textsc{Block})$$

$$\frac{\Gamma; f \vdash t : T \,!\, e}{\Gamma; f \vdash \mathtt{blocking}\ t : T \,!\, \mathit{eff}(\textsc{Blocking}, e)} \qquad (\text{T-}\textsc{Blocking})$$

Finally, to complete the description of the new effect domain we have to define an $\mathit{eff}_{\mathcal{B}}$ function:

$$\begin{aligned}
\mathit{eff}_{\mathcal{B}}(\textsc{Future}, e) &= e \\
\mathit{eff}_{\mathcal{B}}(\textsc{Block}) &= \mathtt{B} \\
\mathit{eff}_{\mathcal{B}}(\textsc{Blocking}, e_1) &= \mathtt{noB}
\end{aligned}$$

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

3.5 Dynamic Semantics

In order to model the runtime behavior of our formal language we define a big-step operational semantics. A term t reduces in one step to either a value v or an error $\mathtt{throw}(p)$, written $t \Downarrow \langle r, S\rangle$ where $r ::= v \mid \mathtt{throw}(p)$. The set S contains the effects that occurred while evaluating the term. Every element $e \in S$ is an atomic effect, i.e., $S \subseteq E$ where E is the effect lattice defined in Section 3.1.

Extensible Effect Domains In the sprit of extensibility to multiple effect domains, the evaluation rules are parametrized by an auxiliary function *dynEff* which computes the effect of evaluating a term based on the effects of its subterms. This function is closely related to the function *eff* used in the typing judgements, but it operates on sets of effects instead of atomic effects. The reason is that in contrast to the static semantics, the operational semantics does not approximate the occurrence of two distinct atomic effects by their join, but keeps both effects in the resulting set S.

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

$$\begin{aligned}
\mathit{dynEff}_{\mathcal{E}}(\textsc{App}, S_1, S_2, S_l) &= S_1 \cup S_2 \cup S_l \\
\mathit{dynEff}_{\mathcal{E}}(\textsc{Throw}(p)) &= \mathtt{throws}(p) \\
\mathit{dynEff}_{\mathcal{E}}(\textsc{Try}, S) &= S \\
\mathit{dynEff}_{\mathcal{E}}(\textsc{Catch}(\overline{p}), S_1, S_2) &= (S_1 \setminus \{\mathtt{throws}(p_i) \mid p_i \in \overline{p}\}) \cup S_2
\end{aligned}$$

In order for the type system to be sound, the *eff* function needs to model *dynEff* conservatively and correctly. This requirement is explained in Section 3.6.

Evaluation Rules We now present the evaluation rules.

$$\frac{t_1 \Downarrow \langle \mathtt{throw}(p), S_1\rangle \quad S = \mathit{dynEff}(\textsc{App}, S_1, \emptyset, \emptyset)}{t_1\ t_2 \Downarrow \langle \mathtt{throw}(p), S\rangle}\ (\text{E-}\textsc{App}\text{-E1})$$

$$\frac{t_1 \Downarrow \langle v_1, S_1\rangle \quad t_2 \Downarrow \langle \mathtt{throw}(p), S_2\rangle \quad S = \mathit{dynEff}(\textsc{App}, S_1, S_2, \emptyset)}{t_1\ t_2 \Downarrow \langle \mathtt{throw}(p), S\rangle}\ (\text{E-}\textsc{App}\text{-E2})$$

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

$$\frac{\begin{array}{c} t_1 \Downarrow \langle (x:T) \mapsto t, S_1 \rangle \qquad t_2 \Downarrow \langle v_2, S_2 \rangle \\ t[v_2/x] \Downarrow \langle r, S_l \rangle \qquad S = \mathit{dynEff}(\textsc{App}, S_1, S_2, S_l) \end{array}}{t_1\ t_2 \Downarrow \langle r, S \rangle} \quad (\text{E-}\textsc{App})$$

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

$$\frac{S = \mathit{dynEff}(\textsc{Throw}(p))}{\texttt{throw}(p) \Downarrow \langle \texttt{throw}(p), S \rangle} \quad (\text{E-}\textsc{Throw})$$

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

$$\frac{\begin{array}{c} t_1 \Downarrow \langle \texttt{throw}(p), S_1 \rangle \qquad p \in \overline{p} \\ t_2 \Downarrow \langle r_2, S_2 \rangle \qquad S_t = \mathit{dynEff}(\textsc{Try}, S_1) \\ S = \mathit{dynEff}(\textsc{Catch}(\overline{p}), S_t, S_2) \end{array}}{\texttt{try}\ t_1\ \texttt{catch}(\overline{p})\ t_2 \Downarrow \langle r_2, S \rangle} \quad (\text{E-}\textsc{Try}\text{-E})$$

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

$$\frac{t_1 \Downarrow \langle r_1, S_1 \rangle \qquad S = \mathit{dynEff}(\textsc{Try}, S_1)}{\texttt{try}\ t_1\ \texttt{catch}(\overline{p})\ t_2 \Downarrow \langle r_1, S \rangle} \quad (\text{E-}\textsc{Try})$$

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

3.6 Effect Soundness

In this section we state two important theorems for the soundness of the type system presented in Section 3.3 with respect to the dynamic semantics introduced in the previous section.

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

Theorem 1. *Preservation.*
If $\Gamma; f \vdash t : T\ !\ e$ is a valid typing statement for term t and the term evaluates as $t \Downarrow \langle r, S \rangle$, then there is valid a typing statement $\Gamma; f \vdash r : T'\ !\ e'$ for r with $T' <: T$.

Theorem 2. *Effect soundness.*
If $\Gamma; f \vdash t : T \ ! \ e$ *and* $t \Downarrow \langle r, S\rangle$*, then* $S \preceq e \sqcup \mathrm{latent}(\Gamma(f))$*.*

The effect soundness theorem states that every effect that occurs when evaluating a term t is represented in the typing derivation for t. Remember that in the typing rule for effect-polymorphic functions, T-Abs-Poly, the argument function f is propagated in the extended environment $\Gamma; f$. Invocations of f are thereafter treated as pure by typing rule T-App-Param.

Therefore, given a typing statement $\Gamma; f \vdash t : T \ ! \ e$, the effect that might occur when evaluating t consists of e *and* the latent effect of f, *latent*$(\Gamma(f))$.

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

Lemma 2. *Consistency.*

- *Let* $S = \mathrm{dynEff}(\textsc{Trm}, \overline{S})$ *be the set of dynamic effects that occur when evaluating a term* t *of the form* Trm*. The list* $\overline{S}$ *contains an effect set for every subterm of* t*.*
- *Let* $\Gamma; f$ *be an environment and* $\overline{e}$ *be a list of static effects such that every effect set in* $\overline{S}$ *is approximated by* $S_i \preceq e_i \sqcup \mathrm{latent}(\Gamma(f))$*.*
- *Then the static effect* $e = \mathrm{eff}(\textsc{Trm}, \overline{e})$ *is a conservative approximation of the effects in* S*, i.e.,* $S \preceq e \sqcup \mathrm{latent}(\Gamma(f))$*.*

This consistency lemma has to be verified for every effect domain. In the case of exceptions or asynchronous operations, the verification is straightforward and therefore omitted here.

Proof Sketch We give a proof sketch for the effect soundness theorem. In addition to preservation, the proof makes use of a lemma showing that effects are preserved in typing statements under value substitution. This lemma comes in two flavors: one for monomorphic and one for effect-polymorphic abstractions.

Lemma 3. *Preservation under substitution for monomorphic abstractions.*
If $\Gamma, x : T_1; f \vdash t : T \ ! \ e_l$*,* $f \neq x$ *and* $\Gamma; g \vdash v : T_2 \ ! \ \bot$ *with* $T_2 <: T_1$*,*
then $\Gamma; f \vdash t[v/x] : T' \ ! \ e'_l$ *such that* $T' <: T$ *and* $e'_l \sqsubseteq e_l$*.*

Lemma 4. *Preservation under substitution for polymorphic abstractions.*
If $\Gamma, x : T_1; x \vdash t : T \ ! \ e_l$ *and* $\Gamma; f \vdash v : T_2 \ ! \ \bot$ *with* $T_2 <: T_1$*,*
then $\Gamma; \epsilon \vdash t[v/x] : T' \ ! \ e'_l$ *such that* $T' <: T$ *and* $e'_l \sqsubseteq e_l \sqcup \mathrm{latent}(T_2)$*.*

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

The proof of Theorem 2 is carried out using induction on the evaluation rules for a term t. We look at the most interesting case E-APP that produces the following derivations:

$t = t_1\ t_2$
$t_1 \Downarrow \langle (x : T_1') \mapsto t_{11}, S_1 \rangle$
$t_2 \Downarrow \langle v_2, S_2 \rangle$
$t_{11}[v_2/x] \Downarrow \langle r, S_l \rangle$
$S = \mathit{dynEff}(\textsc{App}, S_1, S_2, S_l)$

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

$\Gamma; f \vdash t_1 : T_1 \xrightarrow{e_l} T \ !\ e_1$
$\Gamma; f \vdash t_2 : T_2 \ !\ e_2$
$T_2 <: T_1$
$e = \mathit{eff}(\textsc{App}, e_1, e_2, e_l \sqcup \mathit{latent}(T_2))$

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

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

$\Gamma, x : T_1'; x \vdash t_{11} : T' \ !\ e_l' \qquad$ with $T_1 <: T_1',\ T' <: T$ and $e_l' \sqsubseteq e_l$

Applying preservation to the term t_2, we obtain $v_2 : T_2'$ with $T_2' <: T_2$. Using transitivity of subtyping, we obtain $T_2' <: T_1'$, and we can apply the substitution Lemma 4 to obtain

$\Gamma; \epsilon \vdash t_{11}[v_2/x] : T'' \ !\ e_l'' \qquad$ with $T'' <: T'$ and $e_l'' \sqsubseteq e_l' \sqcup \mathit{latent}(T_2')$

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

$S_l \preceq e_l'' \sqcup \mathit{latent}(\Gamma(\epsilon))$
$S_l \preceq e_l' \sqcup \mathit{latent}(T_2') \qquad$ by $e_l'' \sqsubseteq e_l' \sqcup \mathit{latent}(T_2')$ and $\mathit{latent}(\Gamma(\epsilon)) = \bot$

Since $T_2' <: T_2$ we can easily verify that $\mathit{latent}(T_2') \sqsubseteq \mathit{latent}(T_2)$. Together with the induction hypotheses on t_1 and t_2, we now have the necessary conditions to apply the consistency Lemma 2:

$S_1 \preceq e_1 \sqcup \mathit{latent}(\Gamma(f))$
$S_2 \preceq e_2 \sqcup \mathit{latent}(\Gamma(f))$
$S_l \preceq e_l \sqcup \mathit{latent}(T_2)$

We obtain the desired result:

$\mathit{dynEff}(\textsc{App}, S_1, S_2, S_l) \preceq \mathit{eff}(\textsc{App}, e_1, e_2, e_l \sqcup \mathit{latent}(T_2)) \sqcup \mathit{latent}(\Gamma(f))$

The proofs of the remaining cases are conducted in a similar fashion. A full proof for all lemmas and both theorems is available in a separate technical report [19].

4 Lightweight Polymorphic Effects in Scala

We implemented the ideas presented in the previous section as a generic framework for polymorphic effect-checking in the Scala programming language. The implementation comes in the form of a compiler plugin which adds new phases to the compilation pipeline. These phases are executed after the unaltered type-checking transformation and therefore, effect-checking can be seen as a pluggable type system [3].

4.1 Effect-polymorphic Methods

Using the same ideas as in the formal system presented above, we extended Scala with a new syntactic form for defining effect-polymorphic methods. While ordinary methods are defined using the keyword `def`, an effect-polymorphic method is introduced with `fun`:[6]

```
fun h(f: Int => Int): Int = f(1)
```

The method `h` is polymorphic in the effect of its argument. When applying it to a pure function, then that invocation of `h` does not have a side-effect:

```
h(x => x + 1)
```

However, we glossed over one important property of Scala: it is an object-oriented language, and the arguments passed to methods are objects. This is equally true for first-class functions like `f` from the example. Concretely, a unary function in Scala is represented as an object of type `Function1`:

– The trait `Function1` has one abstract method named `apply`:
  ```
  trait Function1[+A, -R] {
    def apply(x: A): R
  }
  ```
– The type `Int => Int` is a shorthand for `Function1[Int, Int]`
– The function literal `x => x + 1` is syntactic sugar for an anonymous class[7]
  ```
  new Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
  }
  ```

This observation raises the question what it means for the method `h` to be effect-polymorphic in its argument `f`, since `f` is an object. The answer is that in the object-oriented case, a method is polymorphic in the effect of the *member methods* of its argument. In the example, method `h` is effect-polymorphic in the `apply` method of its argument `f`.

The definition of effect-polymorphism for the object-oriented case is a slight extension of the definition we used in the formal system: a method can be effect-polymorphic in multiple argument methods. For instance, in

[6] This syntactic adjustment is the only language change that was performed. Alternatively, we could also have left the language unchanged and used an annotation.

[7] Local type inference defines the two type parameters of `Function1`

```
fun m(a: A): B = a.b()
```

the method `m` is effect-polymorphic in all members of `a`, not only in `a.b`. In the same way, if an effect-polymorphic method has multiple parameters, it is effect-polymorphic in all of them.

This extension is however not a fundamental change, because methods cannot be partially applied. This means that when a method is invoked, all the parameters are defined and therefore all the argument methods and their effects are known. The same situation could be simulated in our calculus using tuples.

4.2 Effect Annotations in Scala

In order to annotate the latent effect of a method in Scala, our framework uses standard type annotations on the return type of the method. For instance, the following signature describes a method that might throw an exception:

```
def doIO(file: String): Unit @throws[IOException] = ...
```

Ordinary methods defined with the keyword `def` are impure by default. The method `doIO` therefore allows any side-effects in effect domains other than exceptions. The annotation `@pure` marks a method as pure in all effect domains, like the $\perp$ annotation in Section 3.1.

Effect-polymorphic methods defined using `fun` are pure by default.

One question is how the effect of anonymous functions should be annotated. For instance, the function literal `(x: Int) => x + 1` has type `Function1[Int, Int]`, but this type does not have any effect annotation.

In order to propagate the effect information of anonymous functions using their types, the effect checking framework makes use of refinement types [16]. Concretely, the type of the above function literal is extended to

```
Function1[Int, Int] {
  def apply(x: Int): Int @pure
}
```

To make function types with effects more compact, we are planning to introduce syntactic sugar for the above case, for instance `Int` $\Rightarrow$ `Int @pure`.

4.3 Practical Experience

We verified the expressiveness of our polymorphic effect system by applying it to the Scala collections framework. To illustrate the process of making a library effect-polymorphic, we look at the method `map` which applies an argument function to all elements of a collection. This method is implemented at the root of the collection hierarchy in class `TraversableLike` [17] and shared by all descending collection classes such as lists, maps, sets and vectors.

In essence, the parent collection class `TraversableLike` has one abstract method `foreach` which we make effect-polymorphic by changing `def` to `fun`:

```
fun foreach[U](f: Elem => U): Unit
```

Using this method, the class provides concrete implementations of common collection operations that are shared across all collection types: `filter`, `map`, `flatMap`, `partition`, `forall` and many more. The implementation of method `map` is as follows:

```
trait TraversableLike[+A, +Repr]
  // ...
  fun map[B, That](f: A => B,
                   implicit bf: CanBuildFrom[Repr, B, That]): That = {
    val b: Builder[B, That] = bf.apply(this)
    this.foreach(x => b += f(x))
    b.result
  }
}
```

A detailed explanation of this code can be found in [17]. For every collection type extending `TraversableLike`, the type parameter `A` represents the element type of the collection, and `Repr` is the collection type itself. The method `map` takes the target element type `B` and the target collection type `That` as argument. Allowing `map` to produce a different collection type than the current `Repr` is important in some situations, as explained in [17].

The additional value argument `bf` is called the "builder factory". It is used to obtain a builder object of type `Builder[B, That]`. The builder is a simple buffer which collects elements of type `B` and produces a collection of type `That` with these elements. Note that the builder factory is an implicit argument which does not have to be provided by the programmer when using `map`, instead it is searched and inserted by the compiler.

Using the method `foreach`, every element of the current collection is mapped with the argument function `f` and added to the builder. At the end, the resulting collection is obtained from the builder using `result`.

As previously with `foreach`, the only change we performed to make `map` effect-polymorphic is changing the definition keyword from `def` to `fun`. Now, the method `map` is effect-polymorphic in the argument function `f`, and in all member methods of the builder factory `bf`. However, the class `CanBuildFrom` has only one member method which is annotated pure:

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

Therefore the builder factory does not contribute any effect, and `map` is effect-polymorphic in its argument function `f`.

Remember that effect-polymorphic functions are pure by default. Therefore, the implementation of `map` is not allowed to have any side-effects, except the effect of the argument `f`. If we look for instance at exceptions, purity can be easily verified:

– Invoking `bf.apply` does not have an effect, as already discussed

- Adding elements to the builder and obtaining the final result do not throw exceptions (i.e., the `+=` and `result` methods are pure)
- The effect of calling `foreach` is the effect of the argument function; the function `x => b += f(x)` calls the function `f`, which is allowed by effect-polymorphism

This line of reasoning is applied by the framework for every checked effect domain. Verifying purity with respect to state modifications in this example is less straightforward. However, a recent type-and-effect system for purity [18] is expressive enough to verify this case, and we are working on integrating it into our framework.

If an effect system is added to the framework in which the implementation of `map` is not pure, then the return type of `map` has to be annotated with the corresponding effect.

4.4 Implementing Concrete Effect Domains

We implemented the framework for polymorphic type-and-effect systems presented in this paper as a plugin for the Scala compiler. Extending the framework with a new effect domain is simple, as explained in Section 2.3: the programmer needs to provide

- an implementation of the effect lattice,
- the annotation definitions for annotating latent effects in the source code,
- two methods describing how to serialize and de-serialize the annotations into lattice elements, and
- an implementation of the *eff* function in the form of a abstract syntax tree traverser.

This information suffices the framework to verify a new effect domain.

Exceptions The implementation of an effect system for exceptions was as straightforward as expected. Empirical validation shows that the annotation overhead is greatly reduced compared to the non-polymorphic `throws` clauses found in Java. The work on anchored exceptions [22] gives more detailed insights to research in this area.

There are however a few cases where a static effect analysis cannot compute the possibly thrown exceptions of an expression correctly. Even though these limitations are unrelated to the polymorphic effect checking framework presented in this paper, we still mention them for completeness.

To make an example, assume that the method `head` of a list class throws an exception when the list is empty. If we analyze the method

```
def headOrZero(l: List[Int]) = if (l.isEmpty) 0 else l.head
```

we clearly see that the invocation `l.head` will not throw an exception. However, the effect system does not take conditional control flow into account and will therefore conclude that the expression might throw an exception.

In Java, this specific problem is bypassed by having unchecked exceptions, a class of exception types which are not tracked by the effect system. The `NoSuchElementException` in Java is such an unchecked exception, and therefore the example expression would be treated as pure.

Another solution we are considering is to allow programmers to override the inferred effects by introducing syntax for effect casting. We think that this can prevent programmers from disabling effect checking altogether just because in some situations, the analysis computes an imprecise result.

5 Related Work

Polymorphic effect systems are introduced in [13] as part of the FX programming language [7]. The overhead of having explicit effect and region parameters is overcome by later work on type and effect inference [21]. However, type reconstructions requires the whole program to be known and does not allow modular reasoning about effects. Our approach allows modular effect checking using latent effect annotations and effect-polymorphic method types. It also enables more immediate feedback while developing an application, for instance in an IDE.

The work on anchored exceptions [22] extends the `throws` annotations in Java's checked exceptions to express effect-polymorphism. Instead of listing the concrete exception types that might be thrown, an annotation can take the form `throws like a.m()` where `a` is a parameter of the method. Their system is tied to checked exceptions, however we believe that the main ideas can be equally applied in a generic system. In order to do so, our type system could be extended with dependent types, allowing effect annotations to refer to parameter names.

Marino et al. [14] describe a generic type-and-effect system that can be instantiated to different effect domains. Our effect framework can be seen as an extension of their work with effect polymorphism. There are a few differences however; most importantly they support a tagging system for run-time values. Reconstructing which tags can flow into a function argument uses a whole-program analysis; the authors refer to type qualifier inference [4]. Our system is designed to work modularly on the basis of annotations.

There exist a number of other approaches than type-and-effect systems to delimit the scope of side effects. The most common alternative is monadic encapsulation of effects, which has been shown to be equivalent to effect systems by Wadler et al. in [23]. One aspect of type-and-effect systems shown in Section 3.1 is that combining multiple effect domains is straightforward, while composing monads on the other hand is difficult [11]. Applicative functors [6] are one promising upcoming alternative to monads which facilitate composition.

6 Conclusions and Future Work

We designed an extensible framework for polymorphic effect checking where multiple effect domains can be integrated modularly. Annotating functions as effect-polymorphic is lightweight in syntax and independent of specific effect domains. We implemented the framework for the Scala programming language in the form of a compiler plugin and successfully applied polymorphic effect checking to real-world examples such as the Scala collections library.

We are studying two directions to make the presented type-and-effect system more expressive. First, we investigate how annotations for effect-polymorphism can be generalized using dependent types. This extension allows us to express effect masking behavior in function types, and to denote the behavior of methods more precisely in the object-oriented case by stating which members of the arguments contribute to effect-polymorphism.

As a second extension, we are studying a generalization to support flow-sensitive effects, so that more effect systems can be expressed as a plugin to our framework. One example of a flow-sensitive effect system is the purity analysis presented in [18].

When integrating a type-and-effect system into an existing programming language, there are also practical issues that need consideration. One question is how to handle the interaction with legacy code that does not provide effect annotations. A possible solution is to consider whole-program effect analysis techniques to reconstruct effect annotations for existing libraries.

References

1. Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 63–74, New York, NY, USA, 2008. ACM.
2. Robert L. Bocchino and Vikram S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 306–332, Berlin, Heidelberg, 2011. Springer-Verlag.
3. Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.
4. Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28:1035–1087, November 2006.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
6. Jeremy Gibbons and Bruno C. D. S. Oliveira. The essence of the iterator pattern. In *McBride, Conor, Uustalu, Tarmo (eds), Mathematically-structured functional programming*, 2006.
7. David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Report on the FX programming language. Technical report, 1992.
8. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification (3rd Edition)*. Addison-Wesley Professional, 2005.
9. Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.
10. Anders Hejlsberg. The trouble with checked exceptions. http://www.artima.com/intv/handcuffs.html, 2003.

11. David King and Philip Wadler. Combining monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.
12. Doug Lea. A Java fork/join framework. In *Java Grande*, pages 36–43, 2000.
13. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.
14. Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.
15. Anna Mikhailova and Alexander Romanovsky. *Supporting evolution of interface exceptions*, pages 94–110. Springer-Verlag New York, Inc., New York, NY, USA, 2001.
16. Martin Odersky. The Scala language specification. http://www.scala-lang.org/docu/files/ScalaReference.pdf, 2011.
17. Martin Odersky and Adriaan Moors. Fighting bit rot with types (experience report: Scala collections). In Ravi Kannan and K Narayan Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
18. David Pearce. JPure: A modular purity system for Java. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19861-8_7.
19. Lukas Rytz and Martin Odersky. Lightweight polymorphic effects - proofs. Technical report, EPFL, 2012.
20. Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539, pages 175–189. Springer, 2011.
21. Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
22. Marko van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 455–471, New York, NY, USA, 2005. ACM.
23. Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.