Relative Effect Declarations for Lightweight Effect-Polymorphism

Lukas Rytz and Martin Odersky

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

EPFL-REPORT-175546

Abstract. Type-and-effect systems are a well-studied approach for reasoning about the computational behavior of programs. A major roadblock in adopting effect systems in popular languages is the tradeoff between expressiveness and verbosity. In this technical report, we present a syntactically lightweight but expressive system for annotating effect-polymorphic behavior of functions. The presented system is independent from any specific effect domains and can be embedded in an extensible type-and-effect system.

1 Introduction

In this paper we present an extension of the work by Rytz et al. on a lightweight polymorphic type-and-effect system [1]. The main idea of the previous work is to introduce a new function type specifically for effect-polymorphic functions. While an ordinary function from A to B with effect e has type $A \overset{e}{\Rightarrow} B$, an effect-polymorphic function has type $A \xrightarrow{e} B$. Only higher-order functions can be effect-polymorphic, so in the latter case, the argument type A has to be a function type.

While the latent effect of an ordinary function fully described by the effect annotation e, the effect of a polymorphic function consists of two parts: the concrete effect e and the effect of the argument function of type A.

Effect-polymorphism is obtained by computing the effect of a polymorphic function at call-site. Suppose we have a function f of type $(A \overset{\top}{\Rightarrow} A) \xrightarrow{\bot} A$: it accepts a function with arbitrary effects and has no other effect than the effect of its argument. In general, an invocation of f might have the topmost effect $\top$. However, in the concrete invocation expression `f ((x: A) ⇒ x)`, we see a pure function is passed as argument to f. Since f has the effect of its argument, we conclude that this specific invocation of f is side-effect free.

In this article we generalize the notation of effect-polymorphic functions by introducing dependent types and relative effects. Instead of having a special function type for functions that are polymorphic in the effect of their argument, every function type can now specify in which of its arguments it is effect-polymorphic. The example function f from the previous paragraph is effect-

polymorphic in its argument function. Using dependent types and relative effects, it has type $(g : A \xrightarrow{\top} A) \xrightarrow[g]{\bot} A$: the function type explicitly states that it is effect-polymorphic in g.

Relative effects are more expressive than effect-polymorphic function types in the presence of currying. For example, the following function type cannot be expressed in the previous system: $(f : A \xrightarrow{\top} A) \xrightarrow{\bot} A \xrightarrow[f]{\bot} A$.

The added expressiveness of relative effects is most helpful when applying the type system to languages with multi-argument functions: in this case, the programmer can specify precisely in which of its arguments a function is effect-polymorphic. In the case of object-oriented languages, this mechanism becomes even more crucial: every argument of a method is an object with a potentially large number of member methods. A programmer would like to specify in which of those member methods a function is polymorphic. In the following example, we use the annotation `@l(...)` to specify relative effects:

```
def m(a: A, f: Function[B, C]): C @l(f.apply) = {
  f.apply(a.getB())
}
```

In this example, the method m is effect-polymorphic in the *apply* method of the argument function f, but not in any of the members of a. The type A can have a large number of members with arbitrary side-effects, but only the getter `getB` is actually used in the body of m.

The idea of relative effects is closely related to the work on anchored exceptions [2]. We break up their connection between relative effect declarations and exceptions and present a generic type-and-effect system that allows to use relative effect in multiple effect domains.

2 Formalization

We formalize the type-and-effect system presented in the previous section using a lambda calculus with dependent types. The language syntax is summarized in Figure 1. A dependent function type has the form $(x : T) \xrightarrow[\overline{x}]{e} T$ where e is the latent effect of the function and the list of variables $\overline{x}$ denotes the parameters in which the function is effect-polymorphic.

Note that we introduce a very limited form of dependent types: the only place where a type can refer to a term is the list of parameters $\overline{x}$ in which a function type is effect-polymorphic.

A parameter name that is not used in any relative effect annotation can be omitted, i.e. the type $(x : T_1) \xrightarrow[\overline{x}]{e} T_2$ can be expressed as $T_1 \xrightarrow[\overline{x}]{e} T_2$ if x does not appear freely in $\overline{x}$ or T_2.

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

- $T_1 \underset{\overline{f}}{\rightarrow} T_2$ is equivalent to $T_1 \xrightarrow[\overline{f}]{\bot} T_2$ if $\overline{f}$ is non-empty.

$$
\begin{array}{lll}
t & ::= x & \text{parameter} \\
& \mid t\ t & \text{application} \\
& \mid v & \text{value} \\
v & ::= (x : T) \underset{\overline{x}}{\rightarrow} t & \text{function abstraction} \\
 & & \\
T & ::= (x : T) \overset{e}{\underset{\overline{x}}{\rightarrow}} T & \text{function type} \\
e & ::= \bot \mid \top \mid e \sqcup e & \text{effect annotation} \\
 & & \\
\Gamma & ::= \emptyset \mid \Gamma, x : T & \text{parameter context} \\
\Pi & ::= \emptyset \mid \Pi, x \mapsto x' & \text{parameter mapping}
\end{array}
$$

Fig. 1. Core language syntax

– $T_1 \rightarrow T_2$ is equivalent to $T_1 \overset{\top}{\rightarrow} T_2$

In other words, effect-polymorphic functions are pure by default, while the default effect for monomorphic functions is $\top$.

2.1 Effect Lattice

The effect annotations on function types have to form a semi-lattice consisting of a set of atomic effects, a join operator $\sqcup$ to compute the combination of two effects, and a sub-effect relation $\sqsubseteq$ which compares two effects.

The bottom effect $\bot$ denotes purity, i.e. the absence of effects, and is a sub-effect of every other effect. The top effect $\top$ denotes functions with arbitrary effects, it is a super-effect of every other effect.

2.2 Subtyping

The subtyping relation presented in Figure 2 is computed with respect to an environment $\Gamma; \Pi$ and has the common reflexivity and transitivity properties. The environment Γ maps variables to their types and Π is a map between corresponding parameter names when comparing function types. We will see later how the environment is needed in order to compare relative effects of function types.

We investigate closely the subtyping rule for function types S-Fun which verifies three preconditions:

– the argument types conform in contravariant fashion, $T_1 <: T_1'$,
– the result types conform covaraintly, $T_2' <: T_2$, and
– the subtype has a smaller (absolute and relative) effect, $(e', \overline{x'}) \preceq (e, \overline{x})$.

The relation $(e', \overline{x'}) \preceq (e, \overline{x})$ is used to compare the effects of two function types. It takes absolute and relative effects into account at the same time:

– the absolute effect e' has to a sub-effect of the effect e, and
– every relative effect $f \in \overline{x'}$ has to be covered by the effect pair $(e, \overline{x})$.

$\boxed{\Gamma \vdash T' <: T}$

$$\frac{\Gamma; \emptyset \vdash T' <: T}{\Gamma \vdash T' <: T} \quad \text{(S-Default)}$$

$\boxed{\Gamma; \Pi \vdash T' <: T}$

$$\frac{}{\Gamma; \Pi \vdash T <: T} \quad \text{(S-Refl)} \qquad \frac{\Gamma; \Pi \vdash T' <: S \qquad \Gamma; \Pi \vdash S <: T}{\Gamma; \Pi \vdash T' <: T} \quad \text{(S-Trans)}$$

$$\frac{\Gamma; \Pi \vdash T_1 <: T_1' \qquad (\Gamma, x' : T_1', x : T_1); (\Pi, x' \mapsto x) \vdash T_2' <: T_2 \qquad (\Gamma, x' : T_1'); (\Pi, x' \mapsto x) \vdash (e', \overline{x'}) \preceq (e, \overline{x})}{\Gamma; \Pi \vdash (x' : T_1') \xrightarrow[\overline{x'}]{e'} T_2' <: (x : T_1) \xrightarrow[\overline{x}]{e} T_2} \quad \text{(S-Fun)}$$

$\boxed{\Gamma; \Pi \vdash (e', \overline{x'}) \preceq (e, \overline{x})}$

$$\frac{e' \sqsubseteq e \qquad \forall f \in \overline{x'} \begin{cases} \Pi(f) \in \Pi(\overline{x}) & \text{or} \\ (\Gamma, y : T_a); \Pi \vdash (e_y, \overline{y}) \preceq (e, \overline{x}) & \text{where } \Gamma(f) = (y : T_a) \xrightarrow[\overline{y}]{e_y} T_b \end{cases}}{\Gamma; \Pi \vdash (e', \overline{x'}) \preceq (e, \overline{x})}$$

$\boxed{\Pi(f) \text{ and } \Pi(\overline{x})}$

$$\Pi(f) = \begin{cases} g & \text{if } f \mapsto g \in \Pi \\ f & \text{otherwise} \end{cases} \qquad \Pi(\overline{x}) = \overline{\Pi(x_i)} \text{ for } x_i \in \overline{x}$$

Fig. 2. Subtyping

There are two possibilities how a relative effect f can be covered by $(e, \overline{x})$: either there exists a corresponding relative effect in $\overline{x}$, i.e. $\Pi(f) \in \Pi(\overline{x})$, or f is expanded to the effect of f's function type, which in turn is covered by $(e, \overline{x})$.

We illustrate the subtyping rule S-Fun and the sub-effecting relation $\preceq$ with a number of examples, which will also highlight the role of the subtyping environment $\Gamma; \Pi$.

- $\emptyset; \emptyset \vdash (f : T \to T) \underset{f}{\to} T <: (g : T \to T) \underset{g}{\to} T$

  In order to compare the effects of the two function types, the environment Π is extended with the mapping $f \mapsto g$. The subtyping rule produces the subderivation $(f : T \to T); (f \mapsto g) \vdash (\bot, f) \preceq (\bot, g)$.

  This sub-effecting statement holds because the relative effect f is represented by the relative effect g, i.e. $\Pi(f) = f \in \{\Pi(g)\} = \{f\}$

– $\emptyset;\emptyset \vdash (f : T \xrightarrow{e} T) \underset{f}{\to} T <: (g : T \xrightarrow{\bot} T) \underset{g}{\to} T$

  Subtyping on the argument types holds due to contravariance. Verifying the relative effect f is the same as in the previous example.

  Note that the subtyping relation holds, even though the subtype is a function that might have an effect, while the supertype is always pure. The reasons are discussed in [1].

– $\emptyset;\emptyset \vdash (f : T \xrightarrow{e} T) \to (T \underset{f}{\to} T) <: (g : T \xrightarrow{e} T) \to (T \xrightarrow{e} T)$

  This example shows that a relative effect (f in this case) can be covered by a concrete effect (e). We obtain the following sub-derivation for the effects of the result types:

  $(f : T \xrightarrow{e} T, g : T \xrightarrow{e} T); (f \mapsto g) \vdash (\bot, f) \preceq (e, \emptyset)$

  The relative effect f in the sub-effect is not covered by any relative effect in the super-effect. Therefore, the effect of f is expanded according to environment Γ to $(e, \emptyset)$, and we obtain the following derivation

  $(f : T \xrightarrow{e} T, g : T \xrightarrow{e} T); (f \mapsto g) \vdash (e, \emptyset) \preceq (e, \emptyset)$

  which trivially holds.

Note that a simpler strategy to implement subtyping, in which all relative effects are directly expanded, is unsound. This is shown in the following exmample:

$$
\begin{array}{l}
\texttt{let}\ h : (f : A \to A) \underset{f}{\to} A = (f : A \to A) \underset{f}{\to} \\
\quad \texttt{let}\ g = (m : A \underset{f}{\to} A) \underset{f}{\to} m\ 1 \\
\quad g\ ((a : A) \to \texttt{throw})
\end{array}
$$

When invoking the inner function g, the argument function type has to conform to $A \underset{f}{\to} A$, i.e. we obtain the subtyping derivation $A \xrightarrow{\text{THROW}} A <: A \underset{f}{\to} A$. By expanding the relative effect f, the righthand side type becomes $A \xrightarrow{\top} A$. Therefore the invocation of g in the example would be allowed. This is however unsound: assume we apply the higher-order function h to a pure argument function, such as $h\ (x : A) \to x$. Since h is effect-polymorphic in its argument function, this invocation would be type-checked as being pure, but the THROW effect would still occur.

In general, an absolute effect in the subtype (such as THROW) cannot be covered by a relative effect in the supertype (such as f).

2.3 Typing Rules

Terms are type-checked by a typing statement of the form $\Gamma; \overline{f} \vdash t : T\ !\ e$. The environment Γ maps variables to their types and $\overline{f}$ keeps tracks of function parameters in which the current expression is effect-polymorphic. Figure 3 presents the typing rules and three auxiliary functions which are required to type-check terms:

- The function $latent_{\Gamma;\overline{f}}(T)$ computes the latent effect of type T in environment $\Gamma;\overline{f}$. The latent effect is the effect that might occur when a function of type T is invoked.
- The function $T[x \mapsto T_x]_{\Gamma;\overline{f}}$ replaces all occurrences of variable x in the type T by the effect of type T_x. Recall that the only way a type can reference a variable is in the form of a relative effect. When a variable x gets out of scope, all relative effects x in the type T need to be instantiated to the effect of the type of x.
- The predicate $\Gamma \vdash T$ verifies that type T is well-formed in environment Γ. Specifically, it verifies that all relative effects refer to parameters that are in scope.

The key to explaining the typing rules is to show the role of the parameter environment $\overline{f}$ in $\Gamma;\overline{f} \vdash t : T \,!\, e$. When a parameter f is listed in the environment $\overline{f}$ while type-checking a term t, this means that t is the body of a function which is effect-polymorphic in f. This is illustrated by the following example:

$$\texttt{let}\ h = (f : Int \xrightarrow{e} Int) \underset{f}{\rightarrow} f\ 10$$

Since function h is effect-polymorphic in the parameter f, its body is type-checked with f in the parameter environment:s

$$(f : Int \xrightarrow{e} Int); f \vdash f\ 10$$

The setup of typing environments for function bodies is performed by the typing rule T-Abs. In order to understand why it is necessary to have the parameter environment, we take a look at the type of function h:

$$h : (f : Int \xrightarrow{e} Int) \xrightarrow[f]{\perp} Int$$

The function h has a relative effect f and no concrete effect ($\perp$). In the body of g, the function f with effect e is invoked. However, because the effect of invoking f is already represented in h's type through the relative effect annotation, the latent effect e is ignored.

The parameter environment is required to keep track of functions whose latent effect can be ignored. As we will see later, it is used in the typing rule T-App when computing the latent effect of a function application. Therefore, the full typing statement for the body of h is the following:

$$(f : Int \xrightarrow{e} Int); f \vdash f\ 10 : Int \,!\, \perp$$

The typing rule for parameters is split up in two cases: T-Param-Fun is a special case needed to implement effect-polymorphism[1], while the standard rule T-Param covers the other cases. The reason for having the additional typing rule is best explained by an example: we continue using the same higher-order function h introduced previously. When type-checking the function body, $f\ 10$, the reference to parameter f is type-checked by typing rule T-Param-Fun in the following way:

[1] The precondition $x \in \overline{f} \vee y \notin \overline{y}$ in T-Param-Fun is explained in Section 3.1

$\boxed{\Gamma; \overline{f} \vdash t : T \,!\, e}$

$$\frac{\Gamma(x) = (y : T_1) \xrightarrow[\overline{y}]{e} T_2 \qquad x \in \overline{f} \vee y \notin \overline{y}}{\Gamma; \overline{f} \vdash x : (y : T_1) \xrightarrow[x]{\bot} T_2 \,!\, \bot} \text{(T-Param-Fun)} \qquad \frac{\Gamma(x) = T}{\Gamma; \overline{f} \vdash x : T \,!\, \bot} \text{(T-Param)}$$

$$\frac{(\Gamma, x : T_1); \overline{x} \vdash t : T_2 \,!\, e \qquad \Gamma \vdash (x : T_1) \xrightarrow[\overline{x}]{e} T_2}{\Gamma; \overline{f} \vdash (x : T_1) \xrightarrow[\overline{x}]{} t : (x : T_1) \xrightarrow[\overline{x}]{e} T_2 \,!\, \bot} \text{(T-Abs)}$$

$$\frac{\begin{array}{c}\Gamma; \overline{f} \vdash t_1 : (x : T_1) \xrightarrow[\overline{x}]{e} T \,!\, e_1 \qquad \Gamma \vdash T_2 <: T_1 \\ \Gamma; \overline{f} \vdash t_2 : T_2 \,!\, e_2 \qquad e_p = \bigsqcup_{f \in (\overline{x} \setminus \overline{f})} \mathit{latent}_{\Gamma;\overline{f}}((\Gamma, x : T_2)(f))\end{array}}{\Gamma; \overline{f} \vdash t_1\ t_2 : T[x \mapsto T_2]_{\Gamma;\overline{f}} \,!\, e_1 \sqcup e_2 \sqcup e \sqcup e_p} \text{(T-App)}$$

$\boxed{\mathit{latent}_{\Gamma;\overline{f}}(T)}$

$$\frac{T = (x : T_1) \xrightarrow[\overline{x}]{e} T_2 \qquad e_p = \bigsqcup_{f \in (\overline{x} \setminus \overline{f})} \mathit{latent}_{\Gamma;\overline{f}}((\Gamma, x : T_1)(f))}{\mathit{latent}_{\Gamma;\overline{f}}(T) = e \sqcup e_p}$$

$\boxed{T[x \mapsto T_x]_{\Gamma;\overline{f}}}$

Case $T_x = (z : T_a) \xrightarrow[\overline{z}]{e_z} T_b$. Let $e_p = \mathit{latent}_{\Gamma;\overline{f}}(T_a)$ if $z \in \overline{z}$, $\bot$ otherwise.

$$\frac{T = (y : T_1) \xrightarrow[\overline{y}]{e_y} T_2 \qquad x \in \overline{y}}{T[x \mapsto T_x]_{\Gamma;\overline{f}} = (y : T_1[x \mapsto T_x]_{\Gamma;\overline{f}}) \xrightarrow[(\overline{y} \setminus \{x\}), (\overline{z} \setminus \{z\})]{e_y \sqcup e_z \sqcup e_p} T_2[x \mapsto T_x]_{\Gamma;\overline{f}}}$$

$$\frac{T = (y : T_1) \xrightarrow[\overline{y}]{e_y} T_2 \qquad x \notin \overline{y}}{T[x \mapsto T_x]_{\Gamma;\overline{f}} = (y : T_1[x \mapsto T_x]_{\Gamma;\overline{f}}) \xrightarrow[\overline{y}]{e_y} T_2[x \mapsto T_x]_{\Gamma;\overline{f}}}$$

$\boxed{\Gamma \vdash T}$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \qquad \Gamma, x : T_1 \vdash T_2 \\ \forall f \in \overline{x}\ .\ (\Gamma, x : T_1)(f) = (y : T_a) \xrightarrow[\overline{y}]{e_f} T_b\end{array}}{\Gamma \vdash (x : T_1) \xrightarrow[\overline{x}]{e} T_2}$$

Fig. 3. Typing Rules

$$(f : \mathit{Int} \xrightarrow{e} \mathit{Int}); f \vdash f : \mathit{Int} \xrightarrow[f]{\bot} \mathit{Int} \,!\, \bot$$

Note that the effect in the assigned function type differs from the effect in the type $\Gamma(f)$: the assigned function type has a relative effect f, instead of the actual latent effect e. No information is lost in this transformation: the relative effect f will simply be expanded when computing the latent effect $\mathit{latent}(\mathit{Int} \xrightarrow[f]{\bot} \mathit{Int})$.

However, there is a crucial difference in the way the invocation $f\ 10$ is type-checked: since the invoked function has a relative effect f, but also the enclosing function h is effect-polymorphic in f, the latent effect f can be ignored: it is already expressed in the enclosing function's type as a relative effect. This is achieved in typing rule T-App when expanding the relative effects of the invoked function:

$$e_p = \bigsqcup_{f \in (\overline{x} \setminus \overline{f})} \mathit{latent}_{\Gamma;\overline{f}}((\Gamma, x : T_2)(f))$$

The list $\overline{x}$ denotes the relative effects of the invoked function, while the list $\overline{f}$ denotes the relative effects of the enclosing function. Every relative effect $f \in \overline{x}$ which is also in $\overline{f}$ can be ignored, therefore only the effects in $(\overline{x} \setminus \overline{f})$ are expanded. In the example, we have $\overline{x} = \overline{f} = \{f\}$, and therefore $(\overline{x} \setminus \overline{f}) = \emptyset$ and $e_p = \bot$.

The last crucial ingredient to achieve effect-polymorphism is computing the effect of a function application. As an example, we look at an invocation of the higher-order function h with type $(f : \mathit{Int} \xrightarrow{e} \mathit{Int}) \xrightarrow[f]{\bot} \mathit{Int}$

$$h\ ((x : \mathit{Int}) \to x + 1)$$

We expect this application to be pure, because h is effect-polymorphic and the argument is a pure function. This is achieved by using the actual argument type (i.e. the pure function) instead of the type of parameter f when expanding the relative effect f.

In the typing rule T-App, T_2 is the type of the actual argument for parameter x. When expanding a relative effect f, the type of f is looked up in the environment $(\Gamma, x : T_2)$:

$$e_p = \bigsqcup_{f \in (\overline{x} \setminus \overline{f})} \mathit{latent}_{\Gamma;\overline{f}}((\Gamma, x : T_2)(f))$$

In the example, the relative effect is expanded as follows:

$$e_p = \mathit{latent}_{\Gamma;\overline{f}}((\Gamma, f : (\mathit{Int} \xrightarrow{\bot} \mathit{Int}))(f)) = \mathit{latent}_{\Gamma;\overline{f}}(\mathit{Int} \xrightarrow{\bot} \mathit{Int}) = \bot$$

The last step to consider in typing rule T-App is that the result type of the invoked function can refer to the function's parameter. Simply assigning that result type to the application expression is therefore wrong, because references to the parameter become unbound. For instance, assume we invoke a function g of type $(f : \mathit{Int} \xrightarrow{e} \mathit{Int}) \xrightarrow{\bot} (\mathit{Int} \xrightarrow[f]{} \mathit{Int})$.

$g\ ((x : Int) \to x + 1)$

The type of the above invocation is a function from *Int* to *Int* with the effect of the argument function, i.e. $\bot$. So the relative effect f in the result type of g is replaced by the actual effect of the argument passed into g. This conversion is expressed in the result type of T-App as $T[x \mapsto T_2]$: every relative effect x in the type T is replaced by the effect of the argument type T_2.

2.4 Least upper bounds, greatest lower bounds

The subtyping rules in Section 2 only define the subtyping relation between two types, but they do not specify how the least upper bound (also called the "join") or the greatest lower bound (the "meet") of two types can be computed. In Figure 4 we give an algorithmic procedure to compute these types.

$\boxed{lub_\Pi(T_1, T_2) = T}$

$$lub_\Pi((x : T_1) \xrightarrow[\overline{x}]{e} T_2, (x' : T_1') \xrightarrow[\overline{x'}]{e'} T_2') =$$

$$(x : glb_\Pi(T_1, T_1')) \xrightarrow[\Pi(\overline{x}) \cup (\Pi, x' \mapsto x)(\overline{x'})]{e_1 \sqcup e_2} lub_{(\Pi, x' \mapsto x)}(T_2, T_2')$$

$\boxed{glb_\Pi(T_1, T_2) = T}$

$$glb_\Pi((x : T_1) \xrightarrow[\overline{x}]{e} T_2, (x' : T_1') \xrightarrow[\overline{x'}]{e'} T_2') =$$

$$(x : lub_\Pi(T_1, T_1')) \xrightarrow[\Pi(\overline{x}) \cap (\Pi, x' \mapsto x)(\overline{x'})]{e_1 \sqcap e_2} glb_{(\Pi, x' \mapsto x)}(T_2, T_2')$$

$\boxed{e_1 \sqcap e_2 = e}$

$e_1 \sqcap e_2 = e$ such that $e \sqsubseteq e_1$ and
$e \sqsubseteq e_2$ and
$e' \sqsubseteq e$ $\forall e'$ such that $e' \sqsubseteq e_1$ and $e' \sqsubseteq e_2$

Fig. 4. Least upper bounds, greatest lower bounds

The algorithmic rules for least upper bounds and greatest lower bounds are needed when implementing the type system. For instance, the type of a conditional expression `if` c `then` t_1 `else` t_2 is the least upper bound of the types of t_1 and t_2.

3 Limitations

There are two main limitations related to the expressiveness of the type and effect system introduced in the previous section. This section presents those limitations, discusses their impact in practice and proposes possible solutions.

3.1 Relative effects and polymorphic functions

A relative effect annotation that refers to a parameter which is itself an effect-polymorphic function can lead to an over-approximation when expanding the relative effect. The reason is that a relative effect annotation such as f only tells that function f is invoked, but it does not reveal any information about the argument that is passed to f. However, if f is effect-polymorphic, its effect depends on the effect of the argument. Therefore the largest possible effect has to be assumed if the actual argument is unknown.

We illustrate this limitation in the following example. Function h takes as argument a higher-order function f which is effect polymorphic in its argument g. The implementation of h passes an effect-free increase function to the parameter function f.

$$\texttt{let}\ h = (f : (g : Int \to Int) \xrightarrow[g]{e} Int) \underset{f}{\to} f\ ((x : Int) \to x + 1)$$

The type of $h : (f : (g : Int \to Int) \xrightarrow[g]{e} Int) \underset{f}{\to} Int$ specifies a relative effect f, but it does not encode the fact that f is applied to a pure function. Assume we invoke h in the following way:

$$h\ ((g : Int \to Int) \underset{g}{\to} g\ 2)$$

If we know the implementation of h, we can conclude that the above expression is pure: the function g will be bound to the increase function and there are no side-effects. However, by only looking at the signature of h this information is not available. Therefore, the typing rule T-App will expand h's relative effect as follows

$$e_p = latent_{\Gamma;\overline{f}}((g : Int \to Int) \underset{g}{\to} Int) = \top$$

and assign effect $\top$ to the invocation expression of h.

We believe that this limitation in expressiveness is rarely relevant in practice because we expect functions that are polymorphic in the effect of their effect-polymorphic argument function to be rare. We will verify this claim by implementing the proposed type-and-effect system for the Scala programming language and adding effect annotations to large bodies of code.

Nevertheless, we propose two solutions to overcome the mentioned limitation. Both solutions require further investigation and verification with respect to practicability.

Changed semantics for relative effects The over-approximation explained in the previous section can be overcome by changing the semantics of relative effect annotations. In the presented type-and-effect system, a relative effect annotation f covers the entire effect of f, i.e. the concrete effect of f and also the relative effects of f. The mentioned imprecision stems from the fact the for the relative effects of f, no argument type is known and the worst-possible effect has to be assumed.

The semantics of a relative effect f could therefore be changed to only cover the concrete effect of f, but not its relative effects. In the example, this would impact the computed effects in the following way:

$$\texttt{let}\ h = (f : (g : Int \to Int) \xrightarrow[g]{e} Int) \underset{f}{\to} f\ ((x : Int) \to x + 1)$$

In the *concrete* effect e of invoking f could be ignored, because the enclosing function h is effect-polymorphic. However, in the invocation of f, the relative effects of f would still be included in the overall invocation effect. Concretely, the relative effect g would be expanded. Since the corresponding argument is a pure function (the increase function), the expansion of effect g is $\bot$. So the function type of h becomes $h : (f : (g : Int \to Int) \xrightarrow[g]{e} Int) \underset{f}{\to} Int$.

When computing the effect of the invocation $h\ ((g : Int \to Int) \underset{g}{\to} g\ 2)$, again h's relative effect f has to be expanded. However, now only the concrete effect of the argument passed for f has to be considered

$$\mathit{latent\text{-}concrete}((g : Int \to Int) \underset{g}{\to} Int) = \bot$$

and the invocation of h can be type-checked with effect $\bot$.

Although if this solution works for the specific example, it does not fix the lack of expressiveness of the relative effect declarations, but instead just moves it to a different place. This is illustrated by the following example:

$$\texttt{let}\ h = (f : (g : Int \to Int) \underset{g}{\to} Int) \underset{f}{\to} f\ ((x : Int) \to \texttt{throw})$$

To compute the effect of invoking f, the relative effect g is expanded to `throw`. Therefore, the type of h becomes $(f : (g : Int \to Int) \underset{g}{\to} Int) \xrightarrow[f]{\texttt{throw}} Int$.

If we now apply h to a pure function, such as $h\ ((g : Int \to Int) \to 1)$, we see that the `throw` effect will not occur: the function g is not called. The typing rules however would still assign effect `throw` to the invocation expression.

It is not clear without systematic evaluation which of the two semantics for relative effects is more useable in practice. However, the originally introduced semantics where a relative effect f covers the entire effect of f is more natural, therefore we introduced this semantics as the default. More work is required to give a more qualified answer to this question.

Relative effects with arguments A more systematic way to solve the presented limitation is to enhance the expressiveness of relative effect annotations: concretely, a relative effect annotation could not only mention the called function, but also the argument value passed into the function.

For instance, a function with parameter $f : (g : Int \to Int) \underset{g}{\to} Int$ could have a relative effect $f\ id$ expressing the fact that the parameter function f is applied to the identity function.

In case the argument is a value that is out of scope in the parameter type of the polymorphic function, existential abstraction could be used to express a relative effect $(f\ g)$ `forSome` $\{g : Int \xrightarrow{e} Int\}$: this relative effect indicates that the parameter function f is applied to some argument of type $Int \xrightarrow{e} Int$.

Clearly the type-and-effect system becomes significantly more complex with the addition of arguments to relative effects. Before investigating it in more detail, we will gain experience with the simpler relative effect annotations and verify if their limitations are relevant in practice.

Effect on typing rule T-Param-Fun The limitation in expressiveness is the reason for a slight complication in the typing rule T-Param-Fun: the additional condition $x \in \overline{f} \vee y \notin \overline{y}$ is required to avoid an over-approximation when computing the effect of invoking an effect-polymorphic function. It is illustrated by the following example, note that function m is *not* effect-polymorphic in its argument f:

$$\texttt{let}\ m = (f : (g : Int \to Int) \underset{g}{\to} Int) \to f\ ((x : Int) \to x + 1)$$

The function m does not have a side-effect: the invoked function f is effect-polymorphic and the argument is a pure increase function, so the relative effect g is expanded to $\bot$. However, if the typing rule T-App-Param was applied to parameter f, it would assign it the following type: $(g : Int \to Int) \underset{f}{\to} Int$. The problem here is that this type breaks effect-polymorphism, as explained above: the expansion of the relative effect f is $\top$, no matter what argument is passed to f.

To avoid this over-approximation, the typing rule T-Param-Fun is only applied to effect-polymorphic functions ($y \in \overline{y}$) if the enclosing function is effect-polymorphic in the selected function x ($x \in \overline{f}$). Otherwise, the standard rule T-Param is applied. In the example, function f is assigned the type $(g : Int \to Int) \underset{g}{\to} Int$.

3.2 Relative effects and curried functions

There is a limitation in expressiveness related to the interplay of curried argument functions and relative effects which leads to an unexpected behavior. Assume a higher-order function h which takes a curried function f as argument:

$$\texttt{let}\ h = (f : \mathit{Int} \xrightarrow{\perp} \mathit{Int} \xrightarrow{e} \mathit{Int}) \underset{f}{\rightarrow} f\ 1\ 2$$

Contrary to the intuition, the type of h is $(f : \mathit{Int} \xrightarrow{\perp} \mathit{Int} \xrightarrow{e} \mathit{Int}) \xrightarrow[f]{e} \mathit{Int}$: the effect e is not represented by the relative effect annotation f. The function f a pure function which returns another function of type $\mathit{Int} \xrightarrow{e} \mathit{Int}$. The relative effect annotation f only covers the effect of f, but not the effect of the function returned by f.

The relative effect annotations introduced previously do not allow to specify the effect behavior of h precisely. It is not possible to refer to a function *obtained through a parameter*, relative effects can only refer to parameter functions themselves.

This limitation can be overcome by encoding multi-argument functions using tupled arguments, or by adding functions with multiple arguments to the language. However, even in languages with multi-argument functions such as Scala, the same issue appears in a different form:

$$\texttt{let}\ h = (f : \mathit{Option}[\mathit{Int} \xrightarrow{e} \mathit{Int}]) \underset{f}{\rightarrow} f.\texttt{getOrElse}(\mathit{id})\ 1$$

Similarly to the previous example, the programmer might want to express that h is effect-polymorphic in the function stored inside the optional value f. This is however not possible using the current form of relative effect annotations.

One possible way to solve this issue is to introduce a change to the semantics of relative effect annotations: a relative effect f would not only include the effect of the function f, but also the effect of every function reachable through f.

One concern with this solution is that it over-approximates effects in the case where an effectful function is reachable from a parameter but not actually used in the function body. This issue becomes more relevant in the case of object-oriented programming because there, objects tend to have a large number of member methods, and only a very small subset of the set of reachable methods for one parameter is used in a method body. For instance, the following method

$$\texttt{def}\ m(a : A) : C\ @l(a) = a.\texttt{getB.computeC}()$$

has a relative effect a, which makes m polymorphic in the effect of method `computeC`. However, the object a and all objects reachable from it have a potentially large number of other methods with arbitrary side-effects. The relative effect $@l(a)$ covers the effect of all of these methods, it does not specify that only the effect of `computeC` can actually occur.

We believe that a recursive expansion of relative effects could work in some situations, but leads to unacceptable over-approximation in others. It is also possible to support both kinds of relative effects and let the programmer pick one of the two on a case-by-case basis.

4 Future work

We plan to extend the language for relative effect declarations with effect masking operations. The resulting type and effect system should be expressive enough to describe the masking behavior of operators such as `try − catch`.

Note that the behavior of masking operators can be expressed in the presented system by defining specific typing rules: for instance, by shaping a typing rule T-Try it is possible to compute the effect of a `try − catch` block correctly. Adding masking declarations to the type system it will make it possible to declare the masking behavior in the function type of `try` and avoid a special typing rule.

References

1. Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight Polymorphic Effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, Berlin, Heidelberg, 2012. Springer-Verlag.
2. 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.