Deciding not to Decide

Sound and Complete Effect Inference in the Presence of Higher-Rank Polymorphism

Patrycja Balik, Szymon Jędras, and Piotr Polesiuk

University of Wrocław
{pbalik,sjedras,ppolesiuk}@cs.uni.wroc.pl

Abstract. Type-and-effect systems help the programmer to organize data and computational effects in a program. While for traditional type systems expressive variants with sophisticated inference algorithms have been developed and widely used in programming languages, type-and-effect systems did not yet gain widespread adoption. One reason for this is that type-and-effect systems are more complex and the existing inference algorithms make compromises between expressiveness, intuitiveness, and decidability. In this work, we present an effect inference algorithm for a type-and-effect system with subtyping, expressive higher-rank polymorphism, and intuitive set-like semantics of effects. In order to deal with scoping issues of higher-rank polymorphism, we delay solving of effect constraints by transforming them into formulae of propositional logic. We prove soundness and completeness of our algorithm with respect to a declarative type-and-effect system. All the presented results have been formalized in the Rocq proof assistant, and the algorithm has been successfully implemented in a realistic programming language.

Keywords: Type-and-effect systems · Higher-rank polymorphism · Effect reconstruction · Constraints · Algebraic type scheme.

1 Introduction

Type-and-effect systems [29,50] permit tracking information not just about the data in a program, but also the allowed behavior of computations. This information is especially important in languages with advanced control mechanisms, such as those featuring algebraic effect handlers [47]. Ideally, a programmer-facing type-and-effect system should be ergonomic, expressive, and intuitive.

In terms of ergonomics, one obstacle is that effect information tends to be quite large. Fortunately, this can be remedied with effect reconstruction, where the effect information omitted by the programmer can be automatically inferred. As for expressiveness, a usable effect system almost certainly needs a form of effect polymorphism. Finally, intuitiveness requires that the representation of effects matches the user's understanding of effect information. Many effect systems described in the literature [29,50,39,63,12,35,6] represent this information as a set of possible behaviors. This design choice leads to both an elegant theory and

intuitive semantics of types with effects for the programmer, but is not always easy to implement in practice.

A common approach is to use effect rows [32,25,30,26,56] to approximate sets of effects. Effect rows can be thought of as a list of behaviors, potentially terminated by a polymorphic row variable. This technique can be easily combined with the Hindley-Milner algorithm, which is the golden standard for type reconstruction, and grants some additional expressiveness in the form of ML-polymorphism. The key advantage of using rows is that unification is decidable and unitary [57,49], making the resulting reconstruction algorithm decidable. Moreover, since it is a fairly lightweight addition to the Hindley-Milner algorithm, it composes well with other extensions.

One such extension is rank-N polymorphism, which has been observed by Xie *et al.* [61] to be very useful in the context of algebraic effects. For example, consider a higher-order function that opens a file for the duration of a computation received as its argument. A naive approach could yield us the type $\forall\beta.\,\texttt{Filepath} \to (\texttt{File} \to_{\texttt{IO}\cdot\beta} \texttt{Unit}) \to_{\texttt{IO}\cdot\beta} \texttt{Unit}$. The effect $\texttt{IO} \cdot \beta$ associated with the argument and the entire function says that input/output effects can be performed, as well as any effects described by the polymorphic variable β. However, this type does not guarantee that the file handle will not be used after it is closed. For example, the argument could store the handle in some data structure for later use. Taking inspiration from Haskell's ST monad [60] and the work of Xie *et al.*, we can use rank-2 polymorphism to restrict the scope of the handle, by rewriting the type as $\forall\beta.\,\texttt{Filepath} \to (\forall\alpha.\,\texttt{File}\ \alpha \to_{\alpha\cdot\beta} \texttt{Unit}) \to_{\texttt{IO}\cdot\beta} \texttt{Unit}$. This time, the argument is polymorphic in the effect α, which is associated with a particular file handle. Since all operations on $\texttt{File}\ \alpha$ have the effect α, they cannot be used outside this function.

The considered example glosses over a certain difficulty with using effect rows. We attempt to perform a concatenation of the variables α and β, but a row cannot contain two row variables. One way of salvaging the ability to express this is to make α of a different kind (of atomic effects, which are elements of effect rows). However, if we do that, the unification of rows stops being unitary, somewhat compromising type and effect reconstruction.

The authors of the Flix programming language [34] chose a different approach. Effects in Flix are set-like, with operations like union, intersection and difference available as effect constructors. Such a system is highly expressive, and admits a decidable inference algorithm via boolean unification [35]. However, Flix does not currently support any form of higher-rank polymorphism.

Effect inference for set-like effects was thoroughly studied in the context of static analysis [51,54,38,40,19,42]. Notably, Talpin and Jouvelot [50] proposed an algorithm that transforms the problem of effect inference into a corresponding problem of satisfying subeffect constraints, which, for set-like effects, behave like set inclusion. Following Jouvelot and Gifford [29], their algorithm implements a variant of ML-polymorphism with algebraic type schemes of the form $\forall\alpha_1 \ldots \alpha_n.\,[\Omega]\ \tau$, which store a set of constraints Ω together with universally quantified variables $\alpha_1 \ldots \alpha_n$. This allows for solving constraints involving poly-

morphic variables to be delayed to the point at which the scheme is instantiated. Another interesting observation made in this area by Nielson and Nielson [41] is that subtyping induced by subeffecting does not affect the shape of types. They propose a two-stage approach to type-and-effect inference, where effects are reconstructed once the type shapes are already known from the previous phase. Unfortunately, solutions introduced in this path of research were designed for static analysis internal to the compiler, and not exposed to the user directly. As a result, they had no need to support rank-N polymorphism, which requires programmer-provided annotations.

In this paper, we present a sound and complete effect reconstruction algorithm that needs minimal annotations to aid ergonomics, supports higher-rank polymorphism for expressiveness, and enjoys an intuitive set-like representation of effects. Though none of the previously existing solutions fully satisfy these requirements, we base our work on the results originating in the area of static analysis. Our algorithm is based on the work of Talpin and Jouvelot with its algebraic type schemes, modified to use the two-stage approach. The main difficulty arose from extending these techniques to support rank-N polymorphism.

Widespread implementations of rank-N tend to require some annotations to regain decidability of inference. In those algorithms, whenever higher-rank polymorphism is required, the annotations include, at the very least, both the polymorphic variable *binding*, as well as each *bound occurrence*. In contrast, our algorithm allows for rank-N effect polymorphism where only the quantifiers have to be indicated, while the bound occurrences can be inferred. This means that the annotation of the argument in the scheme can be simplified from $\forall\alpha.\,(\_ \to_\alpha \_) \to_\alpha \_$ to just $\forall\alpha.\,\_$. As most programmers are not used to programming with effect systems at all, requiring minimal amount of programmer-supplied effect information is fairly important for encouraging greater adoption.

Designing an algorithm that supports the explosive mixture of subeffecting and rank-N polymorphism with such minimal effect annotations is challenging due to issues related to ubiquitous variable binding and a lack of the principal type property in our setting. As an example, suppose that we have a function `f` known to be of type $(\texttt{Int} \to_{\texttt{IO}} \texttt{Int}) \to_{\texttt{DB}} \texttt{Int}$. Now, consider the following function definition.

```
let g (h : ∀α. Int →_ Int) = h (f h)
```

Two possible different types for the function `g` are $(\forall\alpha.\,\texttt{Int} \to_\alpha \texttt{Int}) \to_{\texttt{DB}} \texttt{Int}$ or $(\forall\alpha.\,\texttt{Int} \to_{\texttt{IO}} \texttt{Int}) \to_{\texttt{IO·DB}} \texttt{Int}$, but neither is more general than the other. Indeed, no principal type exists for `g` at all.

When further definitions using `g` are added, it may turn out that only one of these solutions is valid for the complete program. Unfortunately, as we are processing the annotation for `h`, that information is not available, and we do not even know whether the wildcard contains α or not. While algebraic type schemes provide a mechanism to delay the solving of some constraints, it is not enough to delay decisions involving higher-rank polymorphic variables, since they are confined to a smaller scope than the entire scheme of a let-bound identifier.

In our solution to this problem we introduce two new mechanisms. First, we introduce effect guards to delay the decision whether a locally-bound variable should appear in a given effect. Second, when leaving the scope of a quantified variable, we transform constraints involving this variable into formulae, which do not contain effect variables, and can therefore be solved later.

Our contributions can be summarized as follows.

- In Section 3 we propose a sound and complete algorithm for effect inference in the presence of higher-rank polymorphism. Our algorithm works even if we replace rank-N with the more general System F polymorphism.
- In Section 4 we show that the problem of effect inference is decidable even when algebraic type schemes are replaced by simple type schemes without subeffect constraints.
- As the considered problem is prone to subtle variable scoping issues, we have taken care to formalize all the presented results in the Rocq proof assistant. In Section 5 we briefly describe the design decisions we made in our development.
- We have successfully implemented the algorithm as a part of our Fram programming language—a realistic programming language with algebraic effects. In Section 6 we summarize the insights that enable such a practical implementation.

The considered type system and proposed algorithm are quite large, so we focus on the most important ideas in the main body of the paper. The full definitions can be found in the appendices. The implementation of Fram can be found at https://github.com/fram-lang/dbl.

2 Simplified Setting

Before we present a full algorithm, let us start with a simpler problem of effect inference in a language with ML-style effect polymorphism, but without any form of rank-N polymorphism. In this section we present a variation of a well-established solution based on the early work of Talpin and Jouvelot [50] that lays down the foundation for our algorithm presented in the later sections. The algorithm presented in this sections differs from the original formulation, as it has been adjusted to better fit our setting and adapted to the two-stage approach proposed by Nielson and Nielson [41].

2.1 Syntax

The syntax of the simplified calculus is presented in Figure 1. The calculus is a standard lambda calculus extended with the standard polymorphic let-binding. Lambda abstractions are annotated with *syntactic types*, which are built from type variables (α^{T}) and arrows ($T_1 \to_E T_2$) annotated with *syntactic effects* (E), which in turn are built from effect variables (α^{E}), the pure effect (ι), joins ($E_1 \cdot E_2$), and wildcards ($\_$). We use the same metavariables α, β, γ for both

$$
\begin{aligned}
e &::= x \mid \lambda x : T.\, e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e && \text{(expressions)}\\
T &::= \alpha^{\mathsf{T}} \mid T \to_E T && \text{(syntactic types)}\\
E &::= \alpha^{\mathsf{E}} \mid \iota \mid E \cdot E \mid \_ && \text{(syntactic effects)}\\
\tau &::= \alpha^{\mathsf{T}} \mid \tau \to_\varepsilon \tau && \text{(types)}\\
\varepsilon &::= \alpha^{\mathsf{E}} \mid \iota \mid \varepsilon \cdot \varepsilon && \text{(effects)}\\
\Omega &::= \overline{\varepsilon <: \varepsilon} && \text{(sets of constraints)}\\
\sigma &::= \forall \Delta^{\mathsf{E}}.\, [\Omega]\ \tau && \text{(type schemes)}
\end{aligned}
$$

Fig. 1. The syntax of the simplified calculus.

type and effect variables, but we distinguish them by a kind annotation. By convention, we omit the kind annotation when it is clear from the context.

Since syntactic effects contain wildcards, we introduce separate syntactic categories of (internal) types and effects used by the type system, with a similar grammar, except without wildcards. Moreover, we consider two internal effects equal if they can be proven equivalent using the laws of an idempotent, commutative monoid with join ($\cdot$) as the monoidal operation, and the pure effect (ι) as the neutral element. This guarantees a set-like semantics of effects. The syntax of constraints and type schemes will be described later in this section.

2.2 Scopes and Substitutions

We have found that in the presence of higher-rank polymorphism, algorithms as well as their metatheories are prone to errors related to variable binding, such as variable escape or variable capture. Therefore, in the technical presentation of this paper as well as in the accompanying formalization we are very precise about the scopes of variables. To do so, we use the functorial approach to representing syntax [21]. This is crucial for our Rocq formalization. While in the paper we take more traditional approach, some issues should be explained in the paper.

First, each syntactic category is in fact a family of sets of terms indexed by sets of variables that are allowed to occur free. For instance, the set of types is in fact a family of sets $\{\mathsf{Type}(A, B)\}_{A,B}$ indexed by a set A of possibly free effect variables, and a set B of possibly free type variables. In order to avoid escaping variables, each time when we write a term (*e.g.,* a type), there is an explicit or implicit object that "binds" these possibly free variables. For example, in case of a typing relation, these variables are bound by the typing environment.

A *substitution* is a function (or tuple of functions) from *all* potentially free variables to the substituted terms. In the case of types, a substitution θ is a pair of functions $\theta_e \colon A_1 \to \mathsf{Effect}(A_2)$ and $\theta_t \colon B_1 \to \mathsf{Type}(A_2, B_2)$ that substitute for effect variables and type variables respectively. Note that the effects and types returned by θ_e and θ_t are themselves indexed by new sets of potentially free variables A_2 and B_2, which need not be related to the domains of the functions, A_1

Effect Matching $\boxed{\Delta \vdash E \sim \varepsilon}$

$$\frac{}{\Delta \vdash \alpha^{\mathsf{E}} \sim \alpha^{\mathsf{E}}} \qquad \frac{}{\Delta \vdash \iota \sim \iota} \qquad \frac{\Delta \vdash E_1 \sim \varepsilon_1 \qquad \Delta \vdash E_2 \sim \varepsilon_2}{\Delta \vdash E_1 \cdot E_2 \sim \varepsilon_1 \cdot \varepsilon_2} \qquad \frac{}{\Delta \vdash \_ \sim \varepsilon}$$

Type Matching $\boxed{\Delta \vdash T \sim \tau}$

$$\frac{}{\Delta \vdash \alpha^{\mathsf{T}} \sim \alpha^{\mathsf{T}}} \qquad \frac{\Delta \vdash T_1 \sim \tau_1 \qquad \Delta \vdash T_2 \sim \tau_2 \qquad \Delta \vdash E \sim \varepsilon}{\Delta \vdash T_1 \to_E T_2 \;\sim\; \tau_1 \to_\varepsilon \tau_2}$$

Fig. 2. Type and effect matching.

and B_1, in any way. We write $\theta^*\tau$ for a capture-avoiding simultaneous substitution for all free variables in τ according to θ.

When we use a term, for instance $\tau \in \mathsf{Type}(A, B)$, in a context where more variables are available, *e.g.,* as an element of $\mathsf{Type}(X \uplus A, B)$, we have to apply a substitution $\iota_2 \colon A \to X \uplus A$ that injects the old variables into a larger set. While in the formalization we keep this precise, in the paper we omit obvious projections, injections, and permutations to reduce noise. Similarly, we omit identity-like parts of substitutions whenever we substitute only for a subset of the available variables. For instance, when substituting ε for α in $\tau \in \mathsf{Type}(\{\alpha\} \uplus A, B)$, we write $\{\alpha \mapsto \varepsilon\}^*\tau$, which resembles the standard notation for substitution[1].

2.3 Declarative Type System

Here we present a type and effect system, which serves as a specification for the effect inference algorithm presented later in this section. We call this system *declarative*, because it doesn't need to be algorithmic (we allow non-syntax directed rules, and guessing data which are not part of the input), but it should be simple and intuitive. We start from some auxiliary relations. Due to space constraints, we omit most of the obvious rules, and focus on presenting the main ideas and the most notable rules. We refer the reader to Appendix A for the full definition of the system.

Type and effect matching. A programmer may omit some effect annotations using wildcard syntactic effects. The declarative system "guesses" the internal effects that should appear in place of these wildcards. We accomplish this with the *effect matching* relation $\Delta \vdash E \sim \varepsilon$ defined in Figure 2. The relation uses a type environment (Δ), which keeps track of the available type and effect variables in the current scope, and says that the syntactic effect E matches the internal effect ε. As mentioned in Section 2.2, we implicitly assume that both syntactic and

[1] We decided to use this bind-like notation ($\{\alpha \mapsto \varepsilon\}^*\tau$) instead of the standard notation ($\tau\{\alpha \mapsto \varepsilon\}$) in order to keep the notation consistent.

internal effects are well-formed in Δ. For instance, in the first rule we implicitly assume that $\alpha^{\mathsf{E}} \in \Delta$, but we omit such premises to avoid clutter.

The definition of effect matching consists of three obvious structural rules, and the last rule saying that a wildcard can be matched by any (well-formed) effect. On top of this relation we define *type matching*. Since we assume that the types were inferred in the previous phase, the definition of this relation consists of structural rules only.

Constraints, subeffecting, and subtyping. The key feature of the presented type system is that it allows for abstracting over subeffecting constraints of the form $\varepsilon_1 <: \varepsilon_2$. Therefore, the *subeffecting relation* (with the judgement of the form $\Delta; \Omega \vdash \varepsilon_1 <: \varepsilon_2$) uses another environment Ω which is a set of abstracted subeffecting constraints. For a fixed Δ and Ω, the subeffecting relation $\Delta; \Omega \vdash \varepsilon_1 <: \varepsilon_2$ is defined as the least preorder on effects containing Ω, with $\cdot$ as the semilattice join, and ι as the least element. The full definition can be found in the Appendix. Using this relation we can also define the constraint set entailment relation $\Delta; \Omega \vdash \Omega'$ by ensuring that subeffecting holds for every constraint in Ω'. Moreover, subeffecting induces subtyping in the standard way: the definition of $\Delta; \Omega \vdash \tau_1 <: \tau_2$ consists of the standard structural rules with contravariant left-hand-side of the arrow type.

Type schemes. As observed by Jouvelot and Gifford, a reasonable effect reconstruction algorithm may be unable to solve generated constraints while examining a let binding [29]. To address this problem, they propose *algebraic type schemes* to be attached to let-bound variables. In their approach a type scheme abstracts constraints that cannot be solved yet, and requires that these constraints are satisfied at the place of the scheme instantiation.

As the last ingredient of the syntax presented in Figure 1 our type schemes follow this path. A type scheme $\forall \Delta^{\mathsf{E}}. [\Omega]\ \tau$ abstracts a set of effect variables Δ^{E} and remembers a set of subeffecting constraints Ω that may use variables bound in Δ^{E}. We write just τ when both Δ^{E} and Ω are empty. Since we assume a two-stage approach to type-and-effect inference, type schemes do not bind type variables: type polymorphism can be handled in the previous phase, and expressed using rank-N polymorphism introduced in Section 3.

Typing relation. The definition of the typing relation is given in Figure 3. In the variable rule, the scheme assigned to a variable is immediately instantiated with any substitution θ that satisfies the constraints attached to the scheme. In the λ-abstraction rule, the variable is annotated with a syntactic type T, so the actual (monomorphic) type τ_1 assigned to the variable must match the annotation. As usual for type-and-effect systems, the λ-abstraction is pure, but the effect of the body is remembered in the arrow type. The application rule is standard. The let-rule resembles the one known from ML-style polymorphism, but deserves additional explanation. First, it allows to implicitly abstract over some effect variables Δ_g^{E} and constraints Ω_g in the first expression. Moreover, while it allows polymorphic definitions, it enforces *purity restriction* [3] in order to

Typing $\boxed{\Delta; \Omega; \Gamma \vdash e : \tau \,/\, \varepsilon}$

$$\frac{\Gamma(x) = \forall \Delta'.\, [\Omega']\, \tau \qquad \Delta; \Omega \vdash \theta^* \Omega'}{\Delta; \Omega; \Gamma \vdash x : \theta^* \tau \,/\, \iota} \qquad \frac{\Delta \vdash T \sim \tau_1 \qquad \Delta; \Omega; \Gamma, x : \tau_1 \vdash e : \tau_2 \,/\, \varepsilon}{\Delta; \Omega; \Gamma \vdash \lambda x : T.\, e : \tau_1 \to_\varepsilon \tau_2 \,/\, \iota}$$

$$\frac{\Delta; \Omega; \Gamma \vdash e_1 : \tau_2 \to_\varepsilon \tau_1 \,/\, \varepsilon \qquad \Delta; \Omega; \Gamma \vdash e_2 : \tau_2 \,/\, \varepsilon}{\Delta; \Omega; \Gamma \vdash e_1\ e_2 : \tau_1 \,/\, \varepsilon}$$

$$\frac{\Delta, \Delta_g^{\mathsf{E}}; \Omega, \Omega_g; \Gamma \vdash e_1 : \tau_1 \,/\, \iota \qquad \Delta; \Omega; \Gamma, x : \forall \Delta_g^{\mathsf{E}}.\, [\Omega_g]\, \tau_1 \vdash e_2 : \tau_2 \,/\, \varepsilon}{\Delta; \Omega; \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \,/\, \varepsilon} \qquad \frac{\Delta; \Omega; \Gamma \vdash e : \tau \,/\, \varepsilon \qquad \Delta; \Omega \vdash \tau <: \tau' \qquad \Delta; \Omega \vdash \varepsilon <: \varepsilon'}{\Delta; \Omega; \Gamma \vdash e : \tau' \,/\, \varepsilon'}$$

Fig. 3. Typing relation.

ensure that the type system is sound. The last rule is the standard subsumption rule, which allows to change the type and effect of an expression to a supertype and supereffect.

It can be shown that the presented type system is sound wrt the standard operational semantics (not presented here) using the standard methods of logical relations [2,5] or progress and preservation [59,31]. However, these results lie outside the scope of this paper, and therefore we focus our attention on the algorithmic effect inference.

2.4 Algorithmic Effect Inference

The effect inference algorithm proposed by Talpin and Jouvelot [50] is a modification of the well-known Hindley-Milner algorithm $\mathcal{W}$ [36]. In the algorithm $\mathcal{W}$ unknown types are represented by unification variables, which may be instantiated by the unification procedure. Similarly, the presented algorithm uses unification variables to represent unknown effects, and such variables may be instantiated by solving subeffecting constraints. While for convenience in our implementation we use a separate syntactic category for unification variables, in the presentation in this paper, as well as in the formalization, we use regular effect variables for that purpose, but we explicitly keep track which variables were generated.

For being more precise, when the algorithm examines some term defined over the set of effect variables Δ, it returns a tuple $(\Delta_g;\ \ldots)$ whose other elements are defined over the set of effect variables $\Delta \uplus \Delta_g$. Let us start with some auxiliary functions, which correspond to the auxiliary relations of the declarative system.

Type and effect matching. Effect matching is realized by the function that translates syntactic effect to the internal representation and is given in Figure 4. The function is defined by structural recursion on syntactic effects, and returns a tuple containing generated effect variables and translated effects. Most cases

```
tr_effect(α) = (∅; α)
tr_effect(ι) = (∅; ι)
tr_effect(_) =
  fresh α in ({α}; α)

tr_effect(E₁ · E₂) =
  let (Δ₁; ε₁) = tr_effect(E₁) in
  let (Δ₂; ε₂) = tr_effect(E₂) in
  (Δ₁ ⊎ Δ₂; ε₁ · ε₂)
```

Fig. 4. Translation of syntactic effects.

are straightforward and just replace each syntactic effect construct with its corresponding internal effect construct. The only interesting case is for wildcards, where the function "guesses" the effect that should be returned. This "guessing" is realized by generating a fresh variable, similarly to how type guessing is implemented in the original algorithm $\mathcal{W}$.

Using effect matching, we define the function `tr_type`, which translates types. The definition consists of structural cases only, so we omit it in our presentation. Precise definition of this, as well as other functions can be found in Appendix B.

Constraints, subeffecting, and subtyping. Subtyping is realized by a function `subtype` that collects subeffecting constraints that need to be satisfied in order to make the relevant subtyping judgment hold. For arrow types this function is defined as follows.

```
subtype(τ₁' →ε₁ τ₁; τ₂' →ε₂ τ₂) =
  let Ω₁ = subtype(τ₂'; τ₁') in
  let Ω₂ = subtype(τ₁; τ₂) in
  Ω₁ ∪ Ω₂ ∪ {ε₁ <: ε₂}
```

Since there are no effect binders in types, the other cases are straightforward and omitted in the presentation.

The algorithm. Now we are in position to present the sound and complete effect inference algorithm for the simplified problem of this section. The algorithm is presented in Figure 5. The function `infer` takes the typing environment and an expression and returns tuple of the form $(\Delta;\ \tau;\ \varepsilon;\ \Omega)$ containing set of generated variables that may appear in other components of the tuple: inferred type, effect, and the set of constraints that should be satisfied to make typing judgement hold. The function is defined by a structural recursion on the expression.

In the variable case, its scheme $(\forall \Delta_1.\, [\Omega_1]\, \tau)$ is instantiated with fresh effect variables. Since variables introduced by opening a binder are always fresh by convention, it is enough to return the set Δ_1 as the generated variables, together with the type τ and the pure effect. In the declarative system, the constraints Ω_1 needs to be satisfied, so the algorithm includes Ω_1 in the returned tuple.

The type annotation for the λ-abstraction argument needs to be translated to the internal representation and this process may generate new variables (Δ_1). The recursive call to the body is performed in the context where these variables are available and may also generate new variables (Δ_2). Both sets Δ_1 and Δ_2 are

```
infer(Γ; x) =
  let (∀Δ1.[Ω1] τ) = Γ(x) in
  (Δ1; τ; ι; Ω1)

infer(Γ; λx:T.e) =
  let (Δ1; τ1) = tr_type(T) in
  let (Δ2; τ2; ε; Ω) = infer(Γ, x:τ1; e) in
  (Δ1 ⊎ Δ2; τ1 →ε τ2; ι; Ω)

infer(Γ; e1 e2) =
  let (Δ1; τ1; ε1; Ω1) = infer(Γ; e1) in
  match τ1 with
  | τa →ε τv =>
    let (Δ2; τ2; ε2; Ω2) = infer(Γ; e2) in
    let Ωs = subtype(τ2; τa) in
    (Δ1 ⊎ Δ2; τv; ε1 · ε2 · ε; Ω1 ∪ Ω2 ∪ Ωs)
  | _ => fail

infer(Γ; let x = e1 in e2) =
  let (Δ1; τ1; ε1; Ω1) = infer(Γ; e1) in
  infer(Γ, x:∀Δ1.[Ω1 ∪ {ε1 <: ι}] τ1; e2)
```

Fig. 5. The effect inference algorithm for the simplified problem.

combined in the final result. The case for function application is standard and as usual for systems with subtyping, allows expressions of smaller type in the argument position. Note that thanks to two-stage approach, subtype-checking generates only subeffecting constraints. These constraints can be propagated in the result tuple, instead of solving them immediately.

Now we focus on the let-binding case that introduces effect-polymorphism into the language. For the simplified calculus it is surprisingly simple: all variables and constraints generated during the first recursive call are abstracted and put into the algebraic type scheme. The constraint set in the scheme is extended with $\varepsilon_1 <: \iota$ in order to ensure that expressions e_1 is pure.

Remark 1. In the presented algorithm, we generalize all the variables generated during type-checking of the body of the let binding. This differs from the original presentation of Talpin-Jouvelot algorithm as well as classical Hindley-Milner algorithm, where only the variables that do not escape through the environment are generalized. We can do so, because (a) our algorithm doesn't return any substitution and (b) all generated constraints can be included in the type scheme, so generated variables have no mean to escape through the environment. In the algorithm presented in the next section, the condition (b) will be no longer true,

so additional step of constraint factorization will be required. Interestingly, the classical Hindley-Milner algorithm can be presented in the style of the algorithm from this section with additional step of factorizing substitution into its kernel and additional renaming. However, the details are outside the scope of this paper.

The presented algorithm is sound and complete with respect to the declarative type system, which can be stated formally by the following two theorems.

Theorem 1 (Soundness). *For the expression e defined over the set of type variables Δ, if* $\texttt{infer}(\Gamma\texttt{; }e) = (\Delta'\texttt{; }\tau\texttt{; }\varepsilon\texttt{; }\Omega)$*, then* $\Delta, \Delta'; \Omega; \Gamma \vdash e : \tau \mathbin{/} \varepsilon$.

Theorem 2 (Completeness). *If* $\Delta; \Omega; \Gamma \vdash e : \tau \mathbin{/} \varepsilon$*, then* $\texttt{infer}(\Gamma\texttt{; }e) = (\Delta'\texttt{; }\tau'\texttt{; }\varepsilon'\texttt{; }\Omega')$ *for some* Δ', τ', ε', *and* Ω'. *Moreover, there exists substitution* θ *that maps effect variables from* Δ' *to effects over* Δ, *such that* $\Delta; \Omega \vdash \theta^*\Omega'$, $\Delta; \Omega \vdash \theta^*\tau' <: \tau$, *and* $\Delta; \Omega \vdash \theta^*\varepsilon' <: \varepsilon$.

3 Effect Inference with Higher-Rank Polymorphism

A major increase in difficulty arises when we introduce two changes into the underlying type system: higher-rank polymorphism announced in the introduction, and restriction of constraints that can be included in type schemes. Both changes are motivated by practical reasons.

Higher-rank polymorphism. Usually, practical type systems with type reconstruction restrict the expressiveness of parametric polymorphism. For instance, the Hindley-Milner type system allows only polymorphic types in the prenex form, and its more liberal extension to rank-N, distinguish between monomorphic types and polymorphic type schemes, and allows polymorphic variables to be instantiated with monomorphic types only. Since we use two-stage approach, we assume that the types are already inferred, so decisions on particular restrictions are left to the designer of the underlying type system (without effects). Such decisions seem to be orthogonal to the problem of effect inference, so for simplicity we extend the system with a general form of polymorphism in the style of System F. We extend the grammar of expressions, syntactic types, and types as follows.

$$
\begin{aligned}
e &::= \ldots \mid \Lambda\alpha^{\mathsf{T}}.\, e \mid \Lambda\alpha^{\mathsf{E}}.\, e \mid e\ [T] \mid e\ [E] && \text{(expressions)}\\
T &::= \ldots \mid \forall\alpha^{\mathsf{T}}.\, T \mid \forall\alpha^{\mathsf{E}}.\, T && \text{(syntactic types)}\\
\tau &::= \ldots \mid \forall\alpha^{\mathsf{T}}.\, \tau \mid \forall\alpha^{\mathsf{E}}.\, \tau && \text{(types)}
\end{aligned}
$$

Types and syntactic types are extended with polymorphic quantifier, that can abstract both type and effect variables. Places where such polymorphic type is introduced or eliminated are explicitly marked in the syntax of expressions by a type or effect λ-abstraction and application. The type application $(e\ [T])$ contains syntactic type, as it contains complete information inferred by a previous type-inference phase and additionally may contain effects provided by the user.

Type Matching $\boxed{\Delta \vdash T \sim \tau}$

$$\frac{\Delta, \alpha^{\mathsf{T}} \vdash T \sim \tau}{\Delta \vdash \forall \alpha^{\mathsf{T}}. T \sim \forall \alpha^{\mathsf{T}}. \tau} \qquad \frac{\Delta, \alpha^{\mathsf{E}} \vdash T \sim \tau}{\Delta \vdash \forall \alpha^{\mathsf{E}}. T \sim \forall \alpha^{\mathsf{E}}. \tau}$$

Subtyping $\boxed{\Delta; \Omega \vdash \tau <: \tau}$

$$\frac{\Delta, \alpha^{\mathsf{T}}; \Omega \vdash \tau_1 <: \tau_2}{\Delta; \Omega \vdash \forall \alpha^{\mathsf{T}}. \tau_1 <: \forall \alpha^{\mathsf{T}}. \tau_2} \qquad \frac{\Delta, \alpha^{\mathsf{E}}; \Omega \vdash \tau_1 <: \tau_2}{\Delta; \Omega \vdash \forall \alpha^{\mathsf{E}}. \tau_1 <: \forall \alpha^{\mathsf{E}}. \tau_2}$$

Typing $\boxed{\Delta; \Gamma; \Omega \vdash e : \tau \;/\; \varepsilon}$

$$\frac{\Delta, \alpha^{\mathsf{T}}; \Omega; \Gamma \vdash e : \tau \;/\; \iota}{\Delta; \Omega; \Gamma \vdash \Lambda \alpha^{\mathsf{T}}. e : \forall \alpha^{\mathsf{T}}. \tau \;/\; \iota} \qquad \frac{\Delta, \alpha^{\mathsf{E}}; \Omega; \Gamma \vdash e : \tau \;/\; \iota}{\Delta; \Omega; \Gamma \vdash \Lambda \alpha^{\mathsf{E}}. e : \forall \alpha^{\mathsf{E}}. \tau \;/\; \iota}$$

$$\frac{\Delta; \Omega; \Gamma \vdash e : \forall \alpha^{\mathsf{T}}. \tau \;/\; \varepsilon \qquad \Delta \vdash T \sim \tau'}{\Delta; \Omega; \Gamma \vdash e\ [T] : \{\alpha \mapsto \tau'\}^* \tau \;/\; \varepsilon} \qquad \frac{\Delta; \Omega; \Gamma \vdash e : \forall \alpha^{\mathsf{E}}. \tau \;/\; \varepsilon \qquad \Delta \vdash E \sim \varepsilon'}{\Delta; \Omega; \Gamma \vdash e\ [E] : \{\alpha \mapsto \varepsilon'\}^* \tau \;/\; \varepsilon}$$

Fig. 6. New inference rules for explicit polymorphism.

Similarly, in the effect application ($e\ [E]$), the syntactic effect may be just a wildcard, but it can be also more precise.

The new constructs come with new inference rules presented in Figure 6. The new matching and subtyping rules are structural, but since they extend the type environment (Δ) in the premise, their presence has serious consequence: wildcards under the quantifier $\forall \alpha^{\mathsf{E}}. T$ can be matched with effects containing α^{E}. The new typing rules are mostly standard, with a minor exception that in the application rules the matching relation is used in order to translate syntactic representation of type or effect to the internal one.

Type schemes revisited. The algebraic type schemes ($\forall \Delta.\ [\Omega]\ \tau$) from the previous section allowed abstracting any set of constraints Ω. While this decision led to a simple, sound, and complete effect inference algorithm, the obtained system is impractical and counter-intuitive. The main reason is that the type scheme may abstract contradictory constraints, *e.g.,* $\mathtt{IO} <: \iota$, delaying reporting programming errors to unexpected moments. As an extreme case, a mistake in a library function may be reported to the user of the library, but not to the developer.

In order to avoid such strange behaviour, we should restrict somehow the constraints that can be included in a type scheme. Such a restriction should (a) disallow contradictory constraints and (b) be intuitive to the programmer. One could say that the Holy Grail would be disallowing constraints at all, and using type scheme of the form $\forall \Delta. \tau$ like in the classical Hindley-Milner type

system. In Section 4 we show that with such a restriction we can still obtain a sound and complete algorithm, but the price we pay seems to be unacceptable.

In this section we propose another restriction. We allow only constraints in the form of upper-bounds of effect variables bound by the scheme. Formally, the scope-aware definition of valid type scheme is the following.

$$\mathsf{UB}_{\Delta}(\Delta') \triangleq \{\alpha <: \varepsilon \mid \alpha \in \Delta' \wedge \varepsilon \in \mathsf{Effect}(\Delta \uplus \Delta')\}$$

$$\mathsf{ValidScheme}(\Delta) \triangleq \{\forall \Delta'.\, [\Omega]\ \tau \mid \forall (\varepsilon_1 <: \varepsilon_2) \in \Omega.\, (\varepsilon_1 <: \varepsilon_2) \in \mathsf{UB}_{\Delta}(\Delta')\}$$

For the purpose of this section we assume that the type schemes assigned to let-bound variables are valid.

Upper-bound constraints are never contradictory, because the are trivially satisfied, when we substitute pure effects for variables bound by the scheme. Moreover, type schemes with upper-bound constraints can give a quite precise information about function behavior, as in the following example.

Example 1. Assume we have function `callLater : (Unit →IO Unit) →DB Unit` that registers given function is some data structure (effect `DB`) for calling it later, where the effect `IO` is available. Consider the following function.

```
let callNowOrLater now f =
  if now then f ()
  else callLater f
```

It can accept the function with the `IO` effect, but there is nothing wrong in passing the pure function. In the latter case, the call to `callNowOrLater` would not perform `IO` effect, so there is no need to pollute its effect with `IO`. It can be achieved by assigning the type scheme

$$\forall \alpha.\, [\alpha <: \mathtt{IO}]\ \mathtt{Bool} \to (\mathtt{Unit} \to_{\alpha} \mathtt{Unit}) \to_{\alpha \cdot \mathtt{DB}} \mathtt{Unit}.$$

Such a general type scheme cannot be expressed in the type system without constraints.

3.1 Effect Guards

Before presenting the algorithm we will try to grab some intuitions that will help us understand the problem we face. Let us go back to one of motivating examples, when the user provided a syntactic type annotation $T \triangleq \forall \alpha^{\mathsf{E}}.\, \beta \to \_\ \beta$. While translating it to the internal representation, a reasonable algorithm would generate a fresh variable γ in place of the wildcard. However, problems arise when it comes to leave the scope of the quantifier. The syntactic type T is matched by both $\forall \alpha^{\mathsf{E}}.\, \beta \to_{\gamma} \beta$ and $\forall \alpha^{\mathsf{E}}.\, \beta \to_{\gamma \cdot \alpha} \beta$, but neither of them is more general than another. The algorithm should decide if it should include α in the effect matched with the wildcard. The idea behind our algorithm is to delay such decisions by conditionally including bound variable in places where it can appear. To do so,

$$
\begin{aligned}
(\alpha^{\mathsf{E}})[\alpha^{\mathsf{E}}] &\triangleq \top & (\beta^{\mathsf{E}})[\alpha^{\mathsf{E}}] &\triangleq \bot \quad (\alpha^{\mathsf{E}} \neq \beta^{\mathsf{E}}) \\
(\iota)[\alpha^{\mathsf{E}}] &\triangleq \bot & (\varepsilon_1 \cdot \varepsilon_2)[\alpha^{\mathsf{E}}] &\triangleq \varepsilon_1[\alpha^{\mathsf{E}}] \vee \varepsilon_2[\alpha^{\mathsf{E}}] \\
(\varepsilon ? \varphi)[\alpha^{\mathsf{E}}] &\triangleq \varepsilon[\alpha^{\mathsf{E}}] \wedge \varphi & \Omega[\alpha^{\mathsf{E}}] &\triangleq \bigwedge_{(\varepsilon_1 <: \varepsilon_2) \in \Omega} \Big(\varepsilon_1[\alpha^{\mathsf{E}}] \Rightarrow \varepsilon_2[\alpha^{\mathsf{E}}]\Big)
\end{aligned}
$$

Fig. 7. Extracting formulae from effects.

we extend the grammar of effects used by the algorithm by the construct $\varepsilon ? \varphi$ called *effect guard*, where φ is a formula of the propositional logic.

$$
\begin{aligned}
\varphi &::= p \mid \top \mid \bot \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi && \text{(formulae)} \\
\varepsilon &::= \ldots \mid \varepsilon ? \varphi && \text{(effects)}
\end{aligned}
$$

The grammar of effects used by the declarative type system remain unchanged. Intuitively, the effect $\varepsilon ? \varphi$ means ε when the formula φ is satisfied, and ι otherwise. With this new construct, the most general type that can be matched with T would be $\forall \alpha^{\mathsf{E}}.\, \beta \to_{\gamma \cdot (\alpha ? p)} \beta$, where p is a fresh propositional variable.

As we extend the syntax of effects, for the purpose of the metatheory of the algorithm we define a version of a declarative type system that take into account the extended grammar of effects. For a given valuation ρ of propositional variables we define matching, subeffecting, subtyping, and typing relations (all denoted with $\vdash_\rho$) all defined by the rules analogous to the original declarative system, with the following three subeffecting rules that handle additional effect guard construct.

$$
\frac{\rho \not\models \varphi}{\Delta; \Omega \vdash_\rho \varepsilon_1 ? \varphi <: \varepsilon_2} \qquad \frac{\Delta; \Omega \vdash_\rho \varepsilon_1 <: \varepsilon_2}{\Delta; \Omega \vdash_\rho \varepsilon_1 ? \varphi <: \varepsilon_2} \qquad \frac{\rho \models \varphi \qquad \Delta; \Omega \vdash_\rho \varepsilon_1 <: \varepsilon_2}{\Delta; \Omega \vdash_\rho \varepsilon_1 <: \varepsilon_2 ? \varphi}
$$

The first rule says that guarded effect is pure, when the formula is false. The second and the third rule say that guard can be discarded if the formula is true. Note that in the second rule, the there is no premise $\rho \models \varphi$ for simplicity: the conclusion always holds when the formula is false, by the first rule.

In order to motivate the second ingredient of our solution, let us consider when the type $\forall \alpha^{\mathsf{E}}.\, \beta \to_{\alpha ? \varphi_1} \beta$ is a subtype of $\forall \alpha^{\mathsf{E}}.\, \beta \to_{\alpha ? \varphi_2} \beta$. The algorithm from Section 2 simply collects subeffecting constraints that are required the subtyping to hold. In this case we cannot simply return the constraint $\alpha ? \varphi_1 <: \alpha ? \varphi_2$, because α would escape its scope. However, we can observe that this constraint is satisfied for each effect substituted for α if and only if the implication $\varphi_1 \Rightarrow \varphi_2$ is true. We can see this implication as another form of constraint, which does not contain effect variables, and can be propagated outside the quantifier.

For a systematic method of generating formula when leaving the scope of the effect variable, we define the operation of extracting formulae guarding this variable in an effect. The definition is given in Figure 7. It simply builds the formula that is true when the effect variable is present in the effect. We extend this definition to (finite) sets of constraints, by taking a conjunction of implications.

3.2 Algorithmic Effect Inference

Once we have intuitive understanding of required tools we proceed to presenting the algorithm. In the presentation we focus mostly on the effect polymorphism, because—as we have seen above—scope leaving requires special attention. On the other hand, type polymorphism doesn't pose similar problems, so we omit it in the presentation. However, we refer interesting reader to the technical appendix for the details of the whole construction.

Type matching. As we observed, the key idea of our algorithm is to delay some decisions by introducing predicates of propositional logic. Therefore, some functions, like `tr_type` generate fresh propositional variables when dealing with problematic effect binders. Similarly to effect variables, we explicitly keep track which propositional variables were generated by the algorithm, so now, the `tr_type` function returns triples of the form $(P;\ \Delta;\ \tau)$, where P is the set of generated propositional variables, while the meaning of other components remains unchanged. For most cases, the set of generated propositional variables is a union of sets generated by subexpressions (and empty when there are no subexpressions). The only interesting case is the following case for effect-polymorphic types.

$$
\begin{aligned}
&\texttt{tr\_type}(\forall \alpha^{\mathsf{E}}.T) = \\
&\quad \texttt{let}\ (P;\ \Delta;\ \tau) = \texttt{tr\_type}(T)\ \texttt{in} \\
&\quad \texttt{fresh}\ P_s = \{p_\beta \mid \beta \in \Delta\}\ \texttt{in} \\
&\quad \texttt{fresh}\ \Delta' = \{\gamma_\beta \mid \beta \in \Delta\}\ \texttt{in} \\
&\quad (P \uplus P_s;\ \Delta';\ \forall \alpha^{\mathsf{E}}.\{\overline{\beta \in \Delta \mapsto \gamma_\beta \cdot \alpha ? p_\beta}\}^* \tau)
\end{aligned}
$$

When leaving the scope of variable α, the algorithm should decide for each effect matched by wildcard if it should contain α. Such effects are represented by effect variables from Δ, so the algorithm delay the decision, by extending each effect containing $\beta \in \Delta$ by $\alpha ? p$. The fresh propositional variable p represents the decision: if it is true, α is included in the effect, otherwise $\alpha ? p$ is pure. This procedure should be done for each effect variable in Δ separately, so we generate fresh p_β for each $\beta \in \Delta$.

Remark 2. This trick works well for the problem effect inferece, because of flat, set-like structure of effects: any effect ε containing variable α is equivalent to $\varepsilon' \cdot \alpha$ for some ε' that doesn't contain α. Similar property doesn't hold for types that have more rigid structure, so we don't see how our method could be applied for the problem of type inference.

Subtyping. The `subtype` function collects constraints that should be satisfied in order for given subtyping judgement to hold. But now, the constraints are represented in two forms: set of subeffecting constraints Ω and propositional-logic formula φ. Therefore, the `subtype` function now returns a pair of the form $(\Omega,\ \varphi)$. As before, the only interesting case concerns the effect-polymorphic quantifier.

$$
\texttt{subtype}(\forall \alpha^{\mathsf{E}}.\tau;\ \forall \alpha^{\mathsf{E}}.\tau') =
$$

```
let (Ω, φ) = subtype(τ, τ′) in
({α ↦ ι}*Ω; φ ∧ Ω[α])
```

Again after returning from the recursive call, we need to prevent the variable α from escaping its scope while preserving the meaning of collected constraints. First, we remove variable α from constraints set Ω, by substituting pure effect for α. The information lost by this step is restored in the form of formula $\Omega[\alpha]$.

Constraint Separation. The other technical challenge in designing the algorithm comes from restricting constraints that can appear in type schemes. When examining let expression, the algorithm should divide constraints from the first subexpressions into two sets: one that can be included in the type scheme, and that should be propagated upwards. With the syntax of effects not extended with guards this task is easy. Any constraint set can be normalized to the set of constraints of the form $\alpha <: \varepsilon$ [40], and from such constraints we can select those that describe upper-bounds for generalized effect variables.

In the case of extended syntax we can proceed similarly, if we slightly relax the upper-bound condition. First, observe that any constraint set can be normalized to set of constraints of the form $\alpha?\varphi <: \varepsilon$ (for details, we refer the reader to the appendix). Then, for the purposes of the algorithm we allow such constraints to appear in a type scheme if α is a variable bound by the scheme. For fixed valuation ρ of propositional variables, such a relaxation doesn't influence on the expressive power of type schemes: if $\rho \models \varphi$ then the constraint $\alpha?\varphi <: \varepsilon$ is equivalent to $\alpha <: \varepsilon$, otherwise it is trivially satisfied.

Now, we can define function that separates constraints as follows.

```
separate(Δ_g; Ω) =
  let Ω_n = normalize(Ω) in
  let Ω_g = {γ?φ <: ε | (γ?φ <: ε) ∈ Ω_n ∧ γ ∈ Δ_g} in
  let Ω_p = {β?φ <: {γ ∈ Δ_g ↦ ι}*ε | (β?φ <: ε) ∈ Ω_n ∧ β ∉ Δ_g} in
  (Ω_g; Ω_p)
```

The function takes set of variables Δ_g that will be generalized in the scheme, and divieds set of constraints Ω into the set Ω_g that will be included in the type scheme, and the set Ω_p of remaining constraints. The `normalize` function normalizes constraints to the form desribed above. Additionally, we substitute pure effects for variables from Δ_g in Ω_p, because Ω_p will be used outside the scope of Δ_g. By doing so, we don't lose any information: all normalized constraints that non-trivially used variables from Δ_g are included in Ω_g.

The algorithm. Now we proceed to discussing the main function `infer` of the reconstruction algorithm. As before, the function takes the environment Γ and the examined expression, but the returned tuple $(P;\ \Delta;\ \tau;\ \varepsilon;\ \Omega;\ \varphi)$ contains two new components: a set P of generated propositional variables, and a formula φ that can be seen as a kind of constraint. Adding these components to the algorithm of Section 2 is straightforward, so we focus on the cases presented in Figure 8, that require deeper explanation. The full algorithm can be found in Appendix B.

```
infer(Γ; Λα^E. e) =
  let (P1; Δ1; τ; ε; Ω1; φ1) = infer(Γ; e) in
  fresh Ps = {pβ | β ∈ Δ1} in
  fresh Δ1' = {γβ | β ∈ Δ1} in
  let θ = {β ∈ Δ1 ↦ γβ · α?pβ} in
  let Ω1' = θ*(Ω1 ∪ {ε <: ι}) in
  (P1 ⊎ Ps; Δ1'; ∀α^E. θ*τ; ι; {α ↦ ι}*Ω1'; φ1 ∧ Ω1'[α])

infer(Γ; e [E]) =
  let (P1; Δ1; τ1; ε1; Ω1; φ1) = infer(Γ; e) in
  match τ1 with
  | ∀α^E. τ =>
    let (Δ2; ε2) = tr_effect(E) in
    (P1; Δ1 ⊎ Δ2; {α ↦ ε2}*τ; ε1; Ω1; φ1)
  | _ => fail

infer(Γ; let x = e1 in e2) =
  let (P1; Δ1; τ1; ε1; Ω1; φ1) = infer(Γ; e1) in
  fresh Δ1' = {βα | α ∈ Δ1} in
  fresh Δg = {γα | α ∈ Δ1} in
  let θ = {α ∈ Δ1 ↦ βα · γα} in
  let (Ωg, Ωp) = separate(Δg; θ*(Ω1 ∪ {ε1 <: ι})) in
  let (P2; Δ2; τ2; ε2; Ω2; φ2) = infer(Γ, x : ∀Δg. [Ωg] θ*τ1; e2) in
  (P1 ⊎ P2; Δ1' ⊎ Δ2; τ2; ε2; Ωp ∪ Ω2; φ1 ∧ φ2)
```

Fig. 8. Selected cases of the effect inference algorithm.

The effect abstraction $\Lambda\alpha^{\mathsf{E}}.\, e$ introduces a new effect variable α, so a special care should be taken when the algorithm leaves the scope of α. First, the algorithm should decide for each variable $\beta \in \Delta_1$ if it should be instantiated with effect containing α or not. We do so the same way as we have seen in the `tr_type` function: by substituting $\gamma \cdot \alpha?p$ for β, where p is a fresh propositional variable representing the decision. Moreover, we should transform constraints Ω_1' to the equivalent form that doesn't contain variable α. We proceed as we have seen in the `subtype` function: by substituting pure effect $\{\alpha \mapsto \iota\}^*\Omega_1'$, and recovering lost information in the form of the formula $\Omega_1'[\alpha]$.

The effect application $e\ [E]$ doesn't pose any problems. In this case, the algorithm makes sure that the type of the expression e is an effect-polymorphic quantifier $\forall\alpha^{\mathsf{E}}.\, \tau$, and just substitutes for α the result of translating syntactic effect E into the internal representation.

For the let-expression, the idea is simple: infer type τ_1 of the first expression, generalize all generated variables in τ_1 together with constraints, and continue

with the second expression. However two additional steps are required: in the first step called *variable splitting*, each variable α from Δ_1 is split into join of two variables β_α and γ_α. The former is propagated as one of generated variables, while the latter is generalized in the polymorphic scheme. In the second step, using the `separate` function constraints are separated into generalizable constraints Ω_g in the relaxed upper-bound form, and remaining contraints Ω_p that are propagated upwards. The purpose of the second step was already explained, when the `separate` was described.

The purpose of variable splitting is to avoid too restrictive constraints produced by the second step. Intuitively, a variable $\alpha \in \Delta_1$ stands for some unknown yet effect. This effect can contain some variables generalized in the scheme (γ_α), as well as other effects defined outside the let-expression. The latter are unknown yet, so they are represented by a variable β_α.

Example 2. To show that variable splitting is needed, consider a set $\{\texttt{IO} <: \alpha\}$ containing single constraint, where α was generated during the type inference. Without variable splitting, the `separate` function would return a pair $(\varnothing;\ \{\texttt{IO} <: \iota\})$, where propagated contraints are obviously contradictory. On the other hand, with variable splitting we obtain a pair $(\varnothing;\ \{\texttt{IO} <: \beta_\alpha\})$ which preserves the original meaning of the input set.

Soundness and Completeness. The presented algorithm is sound and complete with respect to the declarative system from the beginning of the section.

Theorem 3 (Soundness). *If* $\texttt{infer}(\Gamma;\ e) = (P;\ \Delta';\ \tau;\ \varepsilon;\ \Omega;\ \varphi)$ *for expression* e *defined over* Δ*, then* $\Delta, \Delta'; \Omega; \Gamma \vdash_\rho e : \tau \,/\, \varepsilon$ *for each valuation* ρ *satisfying formula* φ*.*

The Soundness Theorem is stated as precise as possible, therefore it uses auxiliary typing relation for the extended syntax. However, knowing the valuation ρ we can simplify all guarded effects and return to the original syntax used by the declarative system.

Corollary 1. *If* $\texttt{infer}(\Gamma;\ e) = (P;\ \Delta';\ \tau;\ \varepsilon;\ \Omega;\ \varphi)$ *for expression* e *defined over* Δ*, then for each valuation* ρ *satisfying formula* φ *there exist* τ'*,* ε'*, and* Ω' *such that* $\Delta, \Delta'; \Omega'; \Gamma \vdash e : \tau' \,/\, \varepsilon'$ *and* $\Delta, \Delta'; \Omega \vdash_\rho \Omega'$*.*

The statement of the completeness says that the algorithm finds the most general solution, *i.e.,* that every derivation of the typing relation must be an instance of the one found by the algorithm.

Theorem 4 (Completeness). *If* $\Delta; \Omega; \Gamma \vdash e : \tau \,/\, \varepsilon$*, then* $\texttt{infer}(\Delta;\ e) = (P';\ \Delta';\ \tau';\ \varepsilon';\ \Omega';\ \varphi)$ *for some* P'*,* Δ'*,* τ'*,* ε'*,* Ω'*, and* φ*. Moreover, there exists substitution* θ *that maps variables from* P' *and* Δ' *to formulae and effects, respectively, defined over* Δ*, such that* $\theta^*\varphi$ *is a tautology,* $\Delta; \Omega \vdash \theta^*\Omega'$*,* $\Delta; \Omega \vdash \theta^*\tau' <: \tau$*, and* $\Delta; \Omega \vdash \theta^*\varepsilon' <: \varepsilon$*.*

3.3 Solving Toplevel Constraints

The algorithm transforms the problem of effect inference to a constraint-solving problem. At the end, we get constraints of two kinds: subeffecting constrainsts Ω and a formula φ. We can solve these constraints in two steps.

1. Since we strictly avoid effect variables escaping their scopes, effect constraints in Ω can contain only top-level effect variables. We can pretend that all top-level effect variables are bound at the begining of the program, so we leave their scope as we proceeded in the effect-abstraction case. We transform the set Ω into formula $\varphi_\Omega \triangleq \bigwedge_{\alpha\in\Delta} \Omega[\alpha]$, where Δ is a set of top-level effects. Note that the scope leaving procedure produces also constranits $\{\alpha \in \Omega \mapsto \iota\}^*\Omega$, but since there are no variables in them, they are trivially satisfied.
2. The remaining work to do is to satisfy the formula $\varphi \wedge \varphi_\Omega$. This is a formula of propositional logic, so we can find a satisfying valuation using *e.g.,* SAT solver.

4 Type System Without Constraints

In this section we consider a slightly modified type system, where type schemes are not allowed to contain constraints.

$$\sigma ::= \forall\Delta.\,\tau$$

Since there is no way of abstracting constraints, the typing, subtyping, and subeffecting judgements don't contain the constraint environment (Ω). Here, we present only the two most notable inference rules of the modified declarative system, as the others remain unchanged (except removing, unnecessary constraint environment). These rules resemble polymorphic instantiation and generalization in the standard declarative formulation of the Hindley-Milner type system [36].

$$\frac{\Gamma(x) = \forall\Delta'.\,\tau}{\Delta;\Gamma \vdash x : \theta^*\tau \;/\; \iota} \qquad \frac{\Delta,\Delta';\Gamma \vdash e_1 : \tau_1 \;/\; \iota \qquad \Delta;\Gamma,x : \forall\Delta'.\,\tau_1 \vdash e_2 : \tau_2 \;/\; \varepsilon}{\Delta;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \;/\; \varepsilon}$$

Interestingly, the problem of effect inference in this system is still decidable. However, the price we pay is that the algorithm generates a huge number of variables, which makes it impractical for real-world programs. Before we go into the details, we start with the two examples that illustrate that the problem is not as simple as it may seem at first glance.

Example 3. Since the let-binding (`let` x `=` e_1 `in` e_2) introduces new effect variables, it might be tempting to abstract a single variable for each effect variable generated during the effect inference of e_1 and leave the scope similarly to the effect-abstraction case from the previous section. The let-binding case in the algorithm would then look like this:

```
infer(Γ; let x = e₁ in e₂) =
  let (P₁; Δ₁; τ₁; ε₁; Ω₁; φ₁) = infer(Γ; e₁) in
```

$$\begin{array}{l}\texttt{fresh}\ \Delta_1' = \{\beta_\alpha \mid \alpha \in \Delta_1\}\ \texttt{in}\\ \texttt{fresh}\ \Delta_g = \{\gamma_\alpha \mid \alpha \in \Delta_1\}\ \texttt{in}\\ \texttt{fresh}\ P_1' = \{p_\alpha \mid \alpha \in \Delta_1\}\ \texttt{in}\\ \texttt{let}\ \theta = \{\overline{\alpha \in \Delta_1 \mapsto \beta_\alpha \cdot \gamma_\alpha ? p_\alpha}\}\ \texttt{in}\\ \texttt{let}\ \Omega_1' = \theta^*(\Omega_1 \cup \{\varepsilon_1 <: \iota\})\ \texttt{in}\\ \texttt{let}\ (P_2;\ \Delta_2;\ \tau_2;\ \varepsilon_2;\ \Omega_2;\ \varphi_2) = \texttt{infer}(\Gamma, x : \forall \Delta_g.\, \theta^*\tau_1;\ e_2)\ \texttt{in}\\ (P_1 \uplus P_1' \uplus P_2; \Delta_1' \uplus \Delta_2;\ \tau_2;\ \varepsilon_2;\ \{\overline{\gamma_\alpha \mapsto \iota}\}^*\Omega_1' \cup \Omega_2;\ \varphi_1 \wedge \varphi_2 \wedge \textstyle\bigwedge_{\gamma \in \Delta_g} \Omega_1'[\gamma]).\end{array}$$

This approach gives us a sound algorithm, but it is not complete. Assume that the algorithm on expression e_1 returns the tuple

$$(P_1;\ \{\alpha_1, \alpha_2\};\ \tau_1(\alpha_1, \alpha_2);\ \iota;\ \{\alpha_1 <: \alpha_2, \alpha_2 <: \alpha_1\};\ \varphi_1),$$

where $\tau_1(\alpha_1, \alpha_2)$ is a type where α_1 and α_2 occur in invariant positions. Since the generated constraints state that α_1 and α_2 are equal, to obtain a completeness the algorithm should assign to variable x a scheme equivalent to $\forall\alpha.\, \tau_1(\alpha, \alpha)$. However, the assigned scheme is

$$\forall \gamma_{\alpha_1}, \gamma_{\alpha_2}.\, \tau_1(\beta_{\alpha_1} \cdot \gamma_{\alpha_1} ? p_{\alpha_1}, \beta_{\alpha_2} \cdot \gamma_{\alpha_2} ? p_{\alpha_2}),$$

and the returned formula contains conjuncts $\theta(\alpha_1)[\gamma_{\alpha_i}] \Rightarrow \theta(\alpha_2)[\gamma_{\alpha_i}]$ for $i = 1, 2$, each of which simplifies to $(\beta_{\alpha_1} \cdot \gamma_{\alpha_1} ? p_{\alpha_1})[\gamma_{\alpha_i}] \Rightarrow (\beta_{\alpha_2} \cdot \gamma_{\alpha_2} ? p_{\alpha_2})[\gamma_{\alpha_i}]$, and further to $p_{\alpha_1} \Rightarrow \bot$ and $p_{\alpha_2} \Rightarrow \bot$. The obtained scheme is not polymorphic at all, so such a naive algorithm is not complete.

From the above example we see that the algorithm should take into account the fact that generated effect variables are not necessarily independent. We can achieve this by allowing any combination of generalized variables (γ) to be substituted for the generated effect variables (α). The change is relatively simple: we substitute for α a join of all the effect variables from Δ_g, guarded by fresh propositional variables. The rest of the algorithm remains unchanged.

$$\begin{array}{l}\texttt{infer}(\Gamma;\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2) =\\ \quad \ldots\\ \quad \texttt{fresh}\ \Delta_g = \ldots\ \texttt{in}\\ \quad \texttt{fresh}\ P_1' = \{p_{\alpha,\gamma} \mid \alpha \in \Delta_1 \wedge \gamma \in \Delta_g\}\ \texttt{in}\\ \quad \texttt{let}\ \theta = \overline{\alpha \in \Delta_1 \mapsto \beta_\alpha \cdot \bullet_{\gamma \in \Delta_g} \gamma ? p_{\alpha,\gamma}}\ \texttt{in}\\ \quad \ldots\end{array}$$

Since we take any combination of variables from Δ_g, the size of Δ_g doesn't have to be the same as the size of Δ_1. The question is how many variables we need to generate in Δ_g to obtain a complete algorithm. The next example shows that sometimes we need to generate strictly more variables than the size of Δ_1.

Example 4. Assume that the size of Δ_g is equal to the size of Δ_1. The above algorithm is sound, but still not complete. To present a counterexample, assume that the algorithm called on subexpression e_1 of let binding $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ returns

$$(P_1;\ \{\alpha_0, \alpha_1, \alpha_2\};\ \tau_1(\alpha_0, \alpha_1, \alpha_2);\ \iota;\ \{\alpha_0 <: \alpha_1 \cdot \alpha_2\};\ \varphi_1),$$

where variables α_0, α_1, and α_2 occur on invariant position in $\tau_1(\alpha_0, \alpha_1, \alpha_2)$. Moreover, suppose that e_2 is typeable only if both

$$\tau_p \stackrel{\triangle}{=} \tau_1(\iota, \iota, \iota) \qquad \tau_i \stackrel{\triangle}{=} \tau_1(A_1 \cdot A_2, A_1 \cdot B_1, A_2 \cdot B_2)$$

are valid instantiations of a type scheme assigned to x (where A_1, A_2, B_1, B_2 are some pairwise different effect constants). The type-scheme assigned to x by the algorithm would be

$$\begin{aligned}\sigma \stackrel{\triangle}{=} \forall\gamma_0, \gamma_1, \gamma_2.\, \tau_1(&\beta_{\alpha_0} \cdot \gamma_0?p_{0,0} \cdot \gamma_1?p_{0,1} \cdot \gamma_2?p_{0,2},\\ &\beta_{\alpha_1} \cdot \gamma_0?p_{1,0} \cdot \gamma_1?p_{1,1} \cdot \gamma_2?p_{1,2},\\ &\beta_{\alpha_2} \cdot \gamma_0?p_{2,0} \cdot \gamma_1?p_{2,1} \cdot \gamma_2?p_{2,2})\end{aligned}$$

and the formula generated from the constraint $\alpha_0 <: \alpha_1 \cdot \alpha_2$ contains a conjunction of the following subformulae.

$$p_{0,0} \Rightarrow p_{1,0} \vee p_{2,0} \qquad p_{0,1} \Rightarrow p_{1,1} \vee p_{2,1} \qquad p_{0,2} \Rightarrow p_{1,2} \vee p_{2,2}$$

An inquisitive reader may verify that there is no valuation of propositional variables that satisfies the above formulae and allows τ_p and τ_i to be valid instantiations of σ. On the other hand, the expression is typeable in the declarative type system if x gets the type scheme

$$\sigma' \stackrel{\triangle}{=} \forall\gamma_0, \gamma_1, \gamma_2, \gamma_3.\, \tau_1(\gamma_0 \cdot \gamma_1, \gamma_0 \cdot \gamma_2, \gamma_1 \cdot \gamma_3).$$

The above example shows that when the size of Δ_g is the same as Δ_1 the algorithm is still not complete. However, it gives us hope that if we are able to find an upper-bound for the number of generalized variables used by the declarative system (in this example four is enough), completeness could be regained. Observe that in a type scheme $\forall\alpha_1, \ldots, \alpha_n.\, \tau$ if $n > 2^{|\tau|}$, where $|\tau|$ is the number of arrows in τ, then there are two variables α_i and α_j that always appear together in τ. We could equate them and get the equivalent scheme $\forall\alpha_1, \ldots \alpha_{i-1}, \alpha_{i+1}, \alpha_n.\, \{\alpha_i \mapsto \alpha_j\}^*\tau$.

Example 5. A type scheme $\forall\beta, \gamma.\, \alpha \to_{\beta\cdot\gamma} \alpha$ is equivalent to $\forall\beta.\, \alpha \to_\beta \alpha$, *i.e.,* they have the same instantiations.

Moreover, $|\tau|$ does not depend on effects (and is invariant with respect to effect substitutions), therefore it is known after the type reconstruction phase! With this intuitions in mind, we can construct the algorithm that is sound and complete with respect to the declarative system and uses $2^{|\tau|}$ as a mentioned upper-bound. In sake of brevity, we present only the let-case.

```
infer(Γ; let x = e1 in e2) =
  let (P1; Δ1; τ1; ε1; Ω1; φ1) = infer(Γ; e1) in
  fresh Δ1' = {β_α | α ∈ Δ1} in
```

$$\begin{array}{l}
\texttt{fresh}\ \ \Delta_g\ \texttt{=}\ \{\gamma_i \mid i \in 1,\ldots,2^{|\tau_1|}\}\ \ \texttt{in}\\
\texttt{fresh}\ \ P_1'\ \texttt{=}\ \{p_{\alpha,\gamma} \mid \alpha \in \Delta_1 \wedge \gamma \in \Delta_g\}\ \ \texttt{in}\\
\texttt{let}\ \ \theta\ \texttt{=}\ \{\overline{\alpha \in \Delta_1 \mapsto \beta_\alpha \cdot \bullet_{\gamma\in\Delta_g}\ \gamma?p_{\alpha,\gamma}}\}\ \ \texttt{in}\\
\texttt{let}\ \ \Omega_1'\ \texttt{=}\ \theta^*(\Omega_1 \cup \{\varepsilon_1 <: \iota\})\ \ \texttt{in}\\
\texttt{let}\ \ (P_2\texttt{;}\ \ \Delta_2\texttt{;}\ \ \tau_2\texttt{;}\ \ \varepsilon_2\texttt{;}\ \ \Omega_2\texttt{;}\ \ \varphi_2)\ \texttt{=}\ \texttt{infer}(\Gamma, x : \forall\Delta_g.\,\theta^*\tau_1\texttt{;}\ \ e_2)\ \ \texttt{in}\\
(P_1 \uplus P_1' \uplus P_2\,;\Delta_1' \uplus \Delta_2\texttt{;}\ \ \tau_2\texttt{;}\ \ \varepsilon_2\texttt{;}\ \ \{\overline{\gamma_\alpha \mapsto \iota}\}^*\Omega_1' \cup \Omega_2\texttt{;}\ \ \varphi_1 \wedge \varphi_2 \wedge \textstyle\bigwedge_{\gamma\in\Delta_g}\Omega_1'[\gamma])\,.
\end{array}$$

Theorem 5 (Soundness). *Suppose that* $\texttt{infer}(\Gamma\texttt{;}\ e) = (P\texttt{;}\ \Delta'\texttt{;}\ \tau\texttt{;}\ \varepsilon\texttt{;}\ \Omega\texttt{;}\ \varphi)$ *for some expression* e *defined over* Δ *and* θ *is a substitution mapping* P *and* Δ, Δ' *to terms over* Δ''*. Then* $\Delta'' \vdash_\rho \theta^*\Omega$ *implies* $\Delta''; \theta^*\Omega; \theta^*\Gamma \vdash_\rho e : \theta^*\tau \,/\, \theta^*\varepsilon$ *for each valuation* ρ *satisfying formula* $\theta^*\varphi$*.*

Theorem 6 (Completeness). *If* $\Delta; \Gamma \vdash e : \tau \,/\, \varepsilon$*, then* $\texttt{infer}(\Delta\texttt{;}\ e) = (P'\texttt{;}\ \Delta'\texttt{;}\ \tau'\texttt{;}\ \varepsilon'\texttt{;}\ \Omega'\texttt{;}\ \varphi)$ *for some* P'*,* Δ'*,* τ'*,* ε'*,* Ω'*, and* φ*. Moreover, there exists substitution* θ *that maps variables from* P' *and* Δ' *to formulae and effects, respectively, defined over* Δ*, such that* $\theta^*\varphi$ *is a tautology,* $\Delta \vdash \theta^*\Omega'$*,* $\Delta \vdash \theta^*\tau' <: \tau$*, and* $\Delta \vdash \theta^*\varepsilon' <: \varepsilon$*.*

5 Formalization

The metatheory of effect inference in presence of general polymorphism is particularly prone to errors related to variable bindings. In order to avoid such mistakes, we formalized all the presented results using Rocq proof assistant. The key issue in such a formalization is the representation of variable binding. We decided to use functorial approach [21] a.k.a. nested datatypes approach [7]. However, the algorithm strongly relies on the finiteness of the generated sets, so parametrizing syntax with arbitrary types instead of finite sets, like in the original nested datatype approach does not work well.

At the beginning we tried to use Binding library [48] designed for a functorial approach, because of its flexibility in the representation of sets of potentially free variables. We parametrized the syntax with a cartesian product of finite sets with a disjoint union as one of constructors. However, Binding heavily uses type classes mechanism that didn't work well when multiple instantiations of the same class were used in a single formalization. Therefore, we ended up with a functorial approach reimplemented from scratch.

6 Implementation and Practical Considerations

Looking at the algorithm presented in this work, it might not be immediately obvious whether it is viable in practice. In the following, we will share some insights we have gained from implementing it in our Fram programming language.

Higher kinds. While our implementation includes arrow kinds $\kappa_1 \to \kappa_2$, it has the restriction that the effect kind cannot appear on the right-hand side of an arrow. Due to this, type-level application cannot appear in effects, and so effects remain simple sets of variables. As we have not found any use cases for effects depending on types or effects in our language, we consider this limitation to be acceptable.

Constraint simplification. The constraints generated by the algorithm can be quite large if left as-is. In turn, the schemes inferred for let definitions run the risk of containing huge constraint sets, which are unwieldy for the programmer to read and inefficient for the implementation to satisfy at each use site of such a definition. Fortunately, in practice, constraints can often be significantly simplified, or completely eliminated, with a collection of straightforward heuristics, for example similar to those in [40].

Checking formula satisfiability. After running our algorithm and eliminating the resulting subeffect constraints, we are left with a propositional-logic formula which needs to be satisfied in order to consider the effect inference successful. Though of course the SAT problem is well-known to be NP-complete, in practice we found that the formulae generated by the algorithm are easy to solve. In the current implementation, we use a hand-written SAT solver that exploits certain properties of the formulae to attain a reasonable runtime, but since many optimized SAT solvers are available, a ready-made solution is also an option.

REPL and incremental solving. When running the interpreter in REPL mode, it is necessary to check the satisfiability of the produced formula for each expression and definition given by the programmer to catch any errors. However, unlike when operating on complete programs, the values of propositional variables cannot be fixed too eagerly, as future input can make some valuations invalid, while other satisfying valuations remain. The simplest implementation can check whether any satisfying valuation exists after each input, but not set the values of any propositional variables. Unfortunately, that results in checking an ever-larger formula every time, which could be a problem for long-running REPL sessions. As an optimization, in our implementation we fix the values of propositional variables as soon as possible if there is only one satisfying valuation for them.

7 Related Work

Effect inference in presence of higher-rank polymorphism. Jouvelot and Gifford [29] presented an algorithm of effect inference in their FX programming language [33,23]. Their algorithm collects effect equivalence constraints and attaches them to algebraic type schemes. Their language supports explicit higher-rank polymorphism and allows omitting type annotation in λ-abstractions, but doesn't support subtyping and wildcards under effect quantifiers. This is the only work that we are aware of that tackles the problem of effect reconstruction in a setup similar to ours. Jouvelot and Gifford claimed that their algorithm is

sound and complete. However, it turned out that they had a subtle bug related to variable binding that made their theorems untrue with no simple fix in sight [28].

Effect inference as a static analysis. After the original work on effect inference by Jouvelot and Gifford, the research community's attention shifted in the direction of static analysis. Talpin and Jouvelot [50] used the technique of algebraic type schemes to perform effect and region inference. Because the region inference is more like a static analysis transparent to the user, considering explicit higher-rank polymorphism doesn't make much sense, so they focused on ML-style polymorphism only, making their work free of mentioned bug. This work started a long line of research on effect based static analysis [51,54,38,52,9,40,19,53,42,24,17]. Nielson and Nielson [39] observed that subtyping doesn't influece shape of types, and concluded that this allows for two-stage approach. Amtoft et al. [1] and Birkedal and Tofte [8] attacked the problem of effect inference where type schemes doesn't contain constraints. However, their systems doesn't admit general polymorhism, and we don't see how their solutions may scale to our system.

Effect inference in surface type systems. The interest in type-and-effect systems as a facility exposed to the programmer has experienced a resurgence alongside the research on algebraic effects [46,47,30,60,4,25,5,63,12,6,61,34,56]. However, the idea is much older, and the previously discussed work by Jouvelot and Gifford [29] fits into this category. Surprisingly, later research in this area does not build on that work, and most of it opts for inference based on row reconstruction [57,49].

The idea to use row polymorphism for effects first appeared in Links [32]. The advantage of this approach is that since row unification is decidable, it is easy to extend the standard Hindley-Milner algorithm with effect rows while maintaining completeness of the inference. Effect rows are essentially lists of behaviors, possibly ending with a row variable. There is a lot of work on effect rows [32,25,30,26,56] with slightly different approaches and design decisions. From our perspective, the main disadvantage of effect rows is that they are conceptually more complex than sets, and also less intuitive. Additionally, some expressiveness is lost by not being able to use multiple polymorphic variables within a row.

Two notable exceptions with effect sets rather than rows include the Flix [34] and Effekt [12] languages. In Flix, effects are sets with all the usual boolean operations such as union, intersection and difference. As a result, effect inference can be implemented using boolean unification [35]. We note the common thread between our solution and the approach used by Flix: both reduce inference to solving boolean algebra problems of some kind, in our case satisfying propositional formulae, and in the case of Flix performing boolean unification on sets. We found the rich grammar of effects in Flix a bit too complicated for our needs, as we sought to present the programmer with effects with just the union operation. Effekt takes an entirely different approach by replacing parametric effect polymorphism with contextual polymorphism. In summary, this view of polymorphism means that effects do not need to contain polymorphic variables at all, and in turn effects can be represented as sets without complicating effect inference. This simplicity comes with a trade-off: functions in Effekt are second-class.

Type inference in System F. Full type reconstruction of System F is known to be undecidable [58], however algorithms to reconstruct types in presence of partial annotation have been developed for some time. Starting with Pierce *et al.* [45], bidirectional approach to this problem became a standard. Here community split into two paths. One dedicated to problem of type checking in System F with explicit type application [15,18,16,11] The other decided to loosen the type system, and adopted rank-N polymorphism [43,44].

Formalization of type reconstruction. Surprisingly, type reconstruction algorithms were rarely formalized using proof assistants for a long time. In the early works [37,14,55,22] a concrete or nominal approach to variable binding was used, and the scopes of variables were managed manually with relatively large overhead. The situation has changed when Dunfield and Krishnaswami [15] proposed a framework for type reconstruction algorithms where scopes of unification variables are tracked via ordered contexts. They didn't formalized their work in a proof assistant, but following their approach many advanced algorithms related to subtyping were formalized in Abela [65,66,64,13], Coq/Rocq [10,27], and Agda [62]. Our result, also explicitly keeps track of variable scopes, but instead of using ordered contexts, we rely on functorial syntax. Recently, Fan *et al.* [20] presented a different approach to formalizing scopes using levels, which is closer to modern implementations of type reconstruction algorithms.

8 Conclusion and Future Work

In this paper we have proposed an effect reconstruction algorithm with set-like semantics of effects and support for higher-rank polymorphism. The algorithm requires only minimal annotations from the programmer. Our algorithm utilizes effect guards and extracts propositional formulae from subeffecting constraints in order to correctly manage the scopes of effect variables. We have proven our algorithm to be both sound and complete with respect to the declarative type-and-effect system. We have also shown that it is feasible to implement in practice by adding it to a realistic programming language with algebraic effects.

This study opens a number of avenues for further research. One concerns the algorithm without constraints in schemes presented in Section 4. Our upper bound for the number of variables needed to preserve the algorithm's completeness is $2^{|\tau|}$. However this estimation does not take into account the number of constraints associated with the variables present in the type. It is easy to see that in the case where there are no such constraints, there is no need to create any fresh variables. This observation suggests that a better upper bound exists, which gives the hope for a practical implementation.

In the opposite direction, we can imagine allowing the programmer to provide algebraic type schemes in rank-N annotations. For example, this would enable types like $\forall\alpha.\,(\forall\beta.\,[\beta <: \alpha]\ \texttt{Int} \to \_\ \texttt{Int}) \to \_\ \texttt{Int}$. We believe it will be possible to have sound and complete algorithm in such a setting, but foresee difficulties associated with calculating the transitive closure of subeffecting constraints.

References

1. Amtoft, T., Nielson, F., Nielson, H.R.: Type and behaviour reconstruction for higher-order concurrent programs. J. Funct. Program. **7**(3), 321–347 (1997), https://doi.org/10.1017/s0956796897002700
2. Appel, A.W., Melliès, P., Richards, C.D., Vouillon, J.: A very modal model of a modern, major, general type system. In: Hofmann, M., Felleisen, M. (eds.) Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17–19, 2007. pp. 109–122. ACM (2007), https://doi.org/10.1145/1190216.1190235
3. Asai, K., Kameyama, Y.: Polymorphic delimited continuations. In: Shao, Z. (ed.) Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29–December 1, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4807, pp. 239–254. Springer (2007), https://doi.org/10.1007/978-3-540-76637-7_16
4. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebr. Methods Program. **84**(1), 108–123 (2015), https://doi.org/10.1016/j.jlamp.2014.02.001
5. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Handle with care: relational interpretation of algebraic effects and handlers. Proc. ACM Program. Lang. **2**(POPL), 8:1–8:30 (2018), https://doi.org/10.1145/3158096
6. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Binders by day, labels by night: effect instances via lexically scoped handlers. PACMPL **4**(POPL), 48:1–48:29 (2020), https://doi.org/10.1145/3371116
7. Bird, R.S., Meertens, L.G.L.T.: Nested datatypes. In: Jeuring, J. (ed.) Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15–17, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1422, pp. 52–67. Springer (1998), https://doi.org/10.1007/BFb0054285
8. Birkedal, L., Tofte, M.: A constraint-based region inference algorithm. Theor. Comput. Sci. **258**(1–2), 299–392 (2001), https://doi.org/10.1016/S0304-3975(00)00025-6
9. Birkedal, L., Tofte, M., Vejlstrup, M.: From region inference to von Neumann machines via region representation inference. In: Boehm, H., Jr., G.L.S. (eds.) Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21–24, 1996. pp. 171–183. ACM Press (1996), https://doi.org/10.1145/237721.237771
10. Bosman, R., Karachalias, G., Schrijvers, T.: No unification variable left behind: Fully grounding type inference for the HDM system. In: Naumowicz, A., Thiemann, R. (eds.) 14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland. LIPIcs, vol. 268, pp. 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023), https://doi.org/10.4230/LIPIcs.ITP.2023.8
11. Botlan, D.L., Rémy, D.: ML^{F}: raising ML to the power of system F. In: Runciman, C., Shivers, O. (eds.) Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003. pp. 27–38. ACM (2003), https://doi.org/10.1145/944705.944709
12. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effects as capabilities: effect handlers and lightweight effect polymorphism. PACMPL **4**(OOPSLA), 126:1–126:30 (2020), https://doi.org/10.1145/3428194

13. Cui, C., Jiang, S., d. S. Oliveira, B.C.: Greedy implicit bounded quantification. Proc. ACM Program. Lang. **7**(OOPSLA2), 2083–2111 (2023), https://doi.org/10.1145/3622871
14. Dubois, C., Ménissier-Morain, V.: Certification of a type inference tool for ML: Damas-Milner within Coq. J. Autom. Reason. **23**(3–4), 319–346 (1999), https://doi.org/10.1023/A:1006285817788
15. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA — September 25 – 27, 2013. pp. 429–442. ACM (2013), https://doi.org/10.1145/2500365.2500582
16. Dunfield, J., Krishnaswami, N.R.: Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. Proc. ACM Program. Lang. **3**(POPL), 9:1–9:28 (2019), https://doi.org/10.1145/3290322
17. Elsman, M.: Explicit effects and effect constraints in ReML. Proc. ACM Program. Lang. **8**(POPL), 2370–2394 (2024), https://doi.org/10.1145/3632921
18. Emrich, F., Lindley, S., Stolarek, J., Cheney, J., Coates, J.: FreezeML: complete and easy type inference for first-class polymorphism. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 423–437. ACM (2020), https://doi.org/10.1145/3385412.3386003
19. Fähndrich, M., Aiken, A.: Program analysis using mixed term and set constraints. In: Hentenryck, P.V. (ed.) Static Analysis, 4th International Symposium, SAS '97, Paris, France, September 8-10, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1302, pp. 114–126. Springer (1997), https://doi.org/10.1007/BFb0032737
20. Fan, A., Xu, H., Xie, N.: Practical type inference with levels. Proc. ACM Program. Lang. **9**(PLDI), 2180–2203 (2025), https://doi.org/10.1145/3729338
21. Fiore, M.P., Plotkin, G.D., Turi, D.: Abstract syntax and variable binding. In: 14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2–5, 1999. pp. 193–202. IEEE Computer Society (1999), https://doi.org/10.1109/LICS.1999.782615
22. Garrigue, J.: A certified implementation of ML with structural polymorphism. In: Ueda, K. (ed.) Programming Languages and Systems — 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 – December 1, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6461, pp. 360–375. Springer (2010), https://doi.org/10.1007/978-3-642-17164-2_25
23. Gifford, D.K., Jouvelot, P., Sheldon, M.A., O'Toole, J.W.: Report on the FX-91 programming language (1992)
24. Helsen, S., Thiemann, P.: Polymorphic specialization for ML. ACM Trans. Program. Lang. Syst. **26**(4), 652–701 (2004), https://doi.org/10.1145/1011508.1011510
25. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: Chapman, J., Swierstra, W. (eds.) Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016. pp. 15–27. ACM (2016), https://doi.org/10.1145/2976022.2976033
26. Ikemori, K., Cong, Y., Masuhara, H., Leijen, D.: Sound and complete type inference for closed effect rows. In: Swierstra, W., Wu, N. (eds.) Trends in Functional Programming — 23rd International Symposium, TFP 2022, Virtual Event, March 17–18, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13401, pp. 144–168. Springer (2022), https://doi.org/10.1007/978-3-031-21314-4_8

27. Jiang, S., Cui, C., d. S. Oliveira, B.C.: Bidirectional higher-rank polymorphism with intersection and union types. Proc. ACM Program. Lang. **9**(POPL), 2118–2148 (2025), https://doi.org/10.1145/3704907
28. Jouvelot, P.: Personal communication
29. Jouvelot, P., Gifford, D.K.: Algebraic reconstruction of types and effects. In: Wise, D.S. (ed.) Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21–23, 1991. pp. 303–310. ACM Press (1991), https://doi.org/10.1145/99583.99623
30. Leijen, D.: Koka: Programming with row polymorphic effect types. In: Levy, P.B., Krishnaswami, N. (eds.) Proceedings of 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, April 12, 2014. EPTCS, vol. 153, pp. 100–126 (2014), https://doi.org/10.4204/EPTCS.153.8
31. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017. pp. 486–499. ACM (2017), https://doi.org/10.1145/3009837.3009872
32. Lindley, S., Cheney, J.: Row-based effect types for database integration. In: Pierce, B.C. (ed.) Proceedings of the 8th ACM SIGPLAN Workshop on Types in Languages Design and Implementation, TLDI 2012, Philadelphia, PA, USA, Saturday, January 28, 2012. pp. 91–102. ACM (2012), https://doi.org/10.1145/2103786.2103798
33. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Ferrante, J., Mager, P. (eds.) Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10–13, 1988. pp. 47–57. ACM Press (1988), https://doi.org/10.1145/73560.73564
34. Madsen, M.: The principles of the Flix programming language. In: Scholliers, C., Singer, J. (eds.) Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand, December 8–10, 2022. pp. 112–127. ACM (2022), https://doi.org/10.1145/3563835.3567661
35. Madsen, M., van de Pol, J.: Polymorphic types and effects with boolean unification. Proc. ACM Program. Lang. **4**(OOPSLA), 154:1–154:29 (2020), https://doi.org/10.1145/3428222
36. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978), https://doi.org/10.1016/0022-0000(78)90014-4
37. Naraschewski, W., Nipkow, T.: Type inference verified: Algorithm W in Isabelle/HOL. J. Autom. Reason. **23**(3-4), 299–318 (1999), https://doi.org/10.1023/A:1006277616879
38. Nielson, F., Nielson, H.R.: Constraints for polymorphic behaviours of concurrent ML. In: Jouannaud, J. (ed.) Constraints in Computational Logics, First International Conference, CCL'94, Munich, Germany, September 7–9, 1994. Lecture Notes in Computer Science, vol. 845, pp. 73–88. Springer (1994), https://doi.org/10.1007/BFb0016845
39. Nielson, F., Nielson, H.R.: Type and effect systems. In: Correct System Design: Recent Insights and Advances, pp. 114–136. Springer (2000)
40. Nielson, F., Nielson, H.R., Amtoft, T.: Polymorphic subtyping for effect analysis: The algorithm. In: Dam, M. (ed.) Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop, Stockholm, Sweden, June 24–26, 1996, Selected Papers. Lecture Notes in Computer Science, vol. 1192, pp. 207–243. Springer (1996), https://doi.org/10.1007/3-540-62503-8_10
41. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999), https://doi.org/10.1007/978-3-662-03811-6

42. Nielson, H.R., Amtoft, T., Nielson, F.: Behaviour analysis and safety conditions: A case study in CML. In: Astesiano, E. (ed.) Fundamental Approaches to Software Engineering, 1st Internationsl Conference, FASE'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 – April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1382, pp. 255–269. Springer (1998), https://doi.org/10.1007/BFb0053595
43. Odersky, M., Läufer, K.: Putting type annotations to work. In: Boehm, H., Jr., G.L.S. (eds.) Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21–24, 1996. pp. 54–67. ACM Press (1996), https://doi.org/10.1145/237721.237729
44. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. JFP **17**(1), 1–82 (2007), https://doi.org/10.1017/S0956796806006034
45. Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. **22**(1), 1–44 (2000), https://doi.org/10.1145/345099.345100
46. Plotkin, G.D., Power, A.J.: Computational effects and operations: An overview. In: Escardó, M., Jung, A. (eds.) Proceedings of the Workshop on Domains VI 2002, Birmingham, UK, September 16–19, 2002. Electronic Notes in Theoretical Computer Science, vol. 73, pp. 149–163. Elsevier (2002), https://doi.org/10.1016/j.entcs.2004.08.008
47. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Log. Methods Comput. Sci. **9**(4) (2013), https://doi.org/10.2168/LMCS-9(4:23)2013
48. Polesiuk, P., Sieczkowski, F.: Functorial syntax for all. In: CoqPL 2024, London, United Kingdom, January 20, 2024 (2024)
49. Rémy, D.: Type inference for records in natural extension of ML, p. 67–95. MIT Press, Cambridge, MA, USA (1994)
50. Talpin, J., Jouvelot, P.: Polymorphic type, region and effect inference. J. Funct. Program. **2**(3), 245–271 (1992), https://doi.org/10.1017/S0956796800000393
51. Talpin, J., Jouvelot, P.: The type and effect discipline. Inf. Comput. **111**(2), 245–296 (1994), https://doi.org/10.1006/inco.1994.1046
52. Tang, Y.M.: Control-Flow Analysis by Effect Systems and Abstract Interpretation. Ph.D. thesis, PhD thesis, Ecoles des Mines de Paris (1994)
53. Tofte, M., Birkedal, L.: A region inference algorithm. ACM Trans. Program. Lang. Syst. **20**(4), 724–767 (1998), https://doi.org/10.1145/291891.291894
54. Tofte, M., Talpin, J.: Implementation of the typed call-by-value lambda-calculus using a stack of regions. In: Boehm, H., Lang, B., Yellin, D.M. (eds.) Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17–21, 1994. pp. 188–201. ACM Press (1994), https://doi.org/10.1145/174675.177855
55. Urban, C., Nipkow, T.: Nominal verification of algorithm W, p. 363–382. Cambridge University Press (2009)
56. de Vilhena, P.E., Pottier, F.: A type system for effect handlers and dynamic labels. In: Wies, T. (ed.) Programming Languages and Systems — 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13990, pp. 225–252. Springer (2023), https://doi.org/10.1007/978-3-031-30044-8_9
57. Wand, M.: Complete type inference for simple objects. In: Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22–25, 1987. pp. 37–44. IEEE Computer Society (1987)

58. Wells, J.B.: Typability and type checking in system F are equivalent and undecidable. Ann. Pure Appl. Log. **98**(1–3), 111–156 (1999), https://doi.org/10.1016/S0168-0072(98)00047-5
59. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994), https://doi.org/10.1006/inco.1994.1093
60. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Swierstra, W. (ed.) Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4–5, 2014. pp. 1–12. ACM (2014), https://doi.org/10.1145/2633357.2633358
61. Xie, N., Cong, Y., Ikemori, K., Leijen, D.: First-class names for effect handlers. Proc. ACM Program. Lang. **6**(OOPSLA2), 30–59 (2022), https://doi.org/10.1145/3563289
62. Xue, X., d. S. Oliveira, B.C.: Contextual typing. Proc. ACM Program. Lang. **8**(ICFP), 880–908 (2024), https://doi.org/10.1145/3674655
63. Zhang, Y., Myers, A.C.: Abstraction-safe effect handlers via tunneling. PACMPL **3**(POPL), 5:1–5:29 (2019), https://doi.org/10.1145/3290318
64. Zhao, J., d. S. Oliveira, B.C.: Elementary type inference. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6–10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 2:1–2:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), https://doi.org/10.4230/LIPIcs.ECOOP.2022.2
65. Zhao, J., d. S. Oliveira, B.C., Schrijvers, T.: Formalization of a polymorphic subtyping algorithm. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving — 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10895, pp. 604–622. Springer (2018), https://doi.org/10.1007/978-3-319-94821-8_36
66. Zhao, J., d. S. Oliveira, B.C., Schrijvers, T.: A mechanical formalization of higher-ranked polymorphic type inference. Proc. ACM Program. Lang. **3**(ICFP), 112:1–112:29 (2019), https://doi.org/10.1145/3341716

A Declarative System

Effect Matching $\boxed{\Delta \vdash E \sim \varepsilon}$

$$\frac{}{\Delta \vdash \alpha^{\mathsf{E}} \sim \alpha^{\mathsf{E}}} \qquad \frac{}{\Delta \vdash \iota \sim \iota} \qquad \frac{\Delta \vdash E_1 \sim \varepsilon_1 \qquad \Delta \vdash E_2 \sim \varepsilon_2}{\Delta \vdash E_1 \cdot E_2 \sim \varepsilon_1 \cdot \varepsilon_2} \qquad \frac{}{\Delta \vdash \_ \sim \varepsilon}$$

Type Matching $\boxed{\Delta \vdash T \sim \tau}$

$$\frac{}{\Delta \vdash \alpha^{\mathsf{T}} \sim \alpha^{\mathsf{T}}} \qquad \frac{\Delta \vdash T_1 \sim \tau_1 \qquad \Delta \vdash T_2 \sim \tau_2 \qquad \Delta \vdash E \sim \varepsilon}{\Delta \vdash T_1 \to_E T_2 \ \sim \ \tau_1 \to_\varepsilon \tau_2}$$

$$\frac{\Delta, \alpha^{\mathsf{T}} \vdash T \sim \tau}{\Delta \vdash \forall \alpha^{\mathsf{T}}. T \sim \forall \alpha^{\mathsf{T}}. \tau} \qquad \frac{\Delta, \alpha^{\mathsf{E}} \vdash T \sim \tau}{\Delta \vdash \forall \alpha^{\mathsf{E}}. T \sim \forall \alpha^{\mathsf{E}}. \tau}$$

Subeffecting $\boxed{\Delta; \Omega \vdash \varepsilon_1 <: \varepsilon_2}$

$$\frac{}{\Delta; \Omega \vdash \varepsilon <: \varepsilon} \qquad \frac{\Delta; \Omega \vdash \varepsilon_1 <: \varepsilon_2 \qquad \Delta; \Omega \vdash \varepsilon_2 <: \varepsilon_3}{\Delta; \Omega \vdash \varepsilon_1 <: \varepsilon_3} \qquad \frac{(\varepsilon_1 <: \varepsilon_2) \in \Omega}{\Delta; \Omega \vdash \varepsilon_1 <: \varepsilon_2}$$

$$\frac{}{\Delta; \Omega \vdash \iota <: \varepsilon} \qquad \frac{\Delta; \Omega \vdash \varepsilon_1 <: \varepsilon \qquad \Delta; \Omega \vdash \varepsilon_2 <: \varepsilon}{\Delta; \Omega \vdash \varepsilon_1 \cdot \varepsilon_2 <: \varepsilon}$$

$$\frac{\Delta; \Omega \vdash \varepsilon <: \varepsilon_1}{\Delta; \Omega \vdash \varepsilon <: \varepsilon_1 \cdot \varepsilon_2} \qquad \frac{\Delta; \Omega \vdash \varepsilon <: \varepsilon_2}{\Delta; \Omega \vdash \varepsilon <: \varepsilon_1 \cdot \varepsilon_2}$$

Subtyping $\boxed{\Delta; \Omega \vdash \tau_1 <: \tau_2}$

$$\frac{}{\Delta; \Omega \vdash \alpha^{\mathsf{T}} <: \alpha^{\mathsf{T}}} \qquad \frac{\Delta; \Omega \vdash \tau_2' <: \tau_1' \qquad \Delta; \Omega \vdash \tau_1 <: \tau_2 \qquad \Delta; \Omega \vdash \varepsilon_1 <: \varepsilon_2}{\Delta; \Omega \vdash \tau_1' \to_{\varepsilon_1} \tau_1 <: \tau_2' \to_{\varepsilon_2} \tau_2}$$

$$\frac{\Delta, \alpha^{\mathsf{T}}; \Omega \vdash \tau_1 <: \tau_2}{\Delta; \Omega \vdash \forall \alpha^{\mathsf{T}}. \tau_1 <: \forall \alpha^{\mathsf{T}}. \tau_2} \qquad \frac{\Delta, \alpha^{\mathsf{E}}; \Omega \vdash \tau_1 <: \tau_2}{\Delta; \Omega \vdash \forall \alpha^{\mathsf{E}}. \tau_1 <: \forall \alpha^{\mathsf{E}}. \tau_2}$$

Typing $\boxed{\Delta; \Gamma; \Omega \vdash e : \tau \ / \ \varepsilon}$

$$\frac{\Gamma(x) = \forall \Delta'.\, [\Omega']\ \tau \qquad \Delta; \Omega \vdash \theta^* \Omega'}{\Delta; \Omega; \Gamma \vdash x : \theta^* \tau \ /\ \iota} \qquad \frac{\Delta \vdash T \sim \tau_1 \qquad \Delta; \Omega; \Gamma, x : \tau_1 \vdash e : \tau_2 \ /\ \varepsilon}{\Delta; \Omega; \Gamma \vdash \lambda x : T.\, e : \tau_1 \to_\varepsilon \tau_2 \ /\ \iota}$$

$$\frac{\Delta, \alpha^{\mathsf{T}}; \Omega; \Gamma \vdash e : \tau \ /\ \iota}{\Delta; \Omega; \Gamma \vdash \Lambda \alpha^{\mathsf{T}}.\, e : \forall \alpha^{\mathsf{T}}.\, \tau \ /\ \iota} \qquad \frac{\Delta, \alpha^{\mathsf{E}}; \Omega; \Gamma \vdash e : \tau \ /\ \iota}{\Delta; \Omega; \Gamma \vdash \Lambda \alpha^{\mathsf{E}}.\, e : \forall \alpha^{\mathsf{E}}.\, \tau \ /\ \iota}$$

$$\frac{\Delta; \Omega; \Gamma \vdash e_1 : \tau_2 \to_\varepsilon \tau_1 \ /\ \varepsilon \qquad \Delta; \Omega; \Gamma \vdash e_2 : \tau_2 \ /\ \varepsilon}{\Delta; \Omega; \Gamma \vdash e_1\ e_2 : \tau_1 \ /\ \varepsilon}$$

$$\frac{\Delta; \Omega; \Gamma \vdash e : \forall \alpha^{\mathsf{T}}.\, \tau \ /\ \varepsilon \qquad \Delta \vdash T \sim \tau'}{\Delta; \Omega; \Gamma \vdash e\ [T] : \{\alpha \mapsto \tau'\}^* \tau \ /\ \varepsilon}$$

$$\frac{\Delta; \Omega; \Gamma \vdash e : \forall \alpha^{\mathsf{E}}.\, \tau \ /\ \varepsilon \qquad \Delta \vdash E \sim \varepsilon'}{\Delta; \Omega; \Gamma \vdash e\ [E] : \{\alpha \mapsto \varepsilon'\}^* \tau \ /\ \varepsilon}$$

$$\frac{\Delta, \Delta_g^{\mathsf{E}}; \Omega, \Omega_g; \Gamma \vdash e_1 : \tau_1 \ /\ \iota \qquad \Delta; \Omega; \Gamma, x : \forall \Delta_g^{\mathsf{E}}.\, [\Omega_g]\ \tau_1 \vdash e_2 : \tau_2 \ /\ \varepsilon}{\Delta; \Omega; \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \ /\ \varepsilon}$$

$$\frac{\Delta; \Omega; \Gamma \vdash e : \tau \ /\ \varepsilon \qquad \Delta; \Omega \vdash \tau <: \tau' \qquad \Delta; \Omega \vdash \varepsilon <: \varepsilon'}{\Delta; \Omega; \Gamma \vdash e : \tau' \ /\ \varepsilon'}$$

B The Algorithm

```
tr_effect(α) = (∅; α)
tr_effect(ι) = (∅; ι)
tr_effect(E₁ · E₂) =
  let (Δ₁; ε₁) = tr_effect(E₁) in
  let (Δ₂; ε₂) = tr_effect(E₂) in
  (Δ₁ ⊎ Δ₂; ε₁ · ε₂)
tr_effect(_) =
  fresh α in
  ({α}; α)

tr_type(α) = (∅; ∅; α)
tr_type(∀α^T.T) =
  let (P; Δ; τ) = tr_type(T) in
  (P; Δ; ∀α^T.τ)

tr_type(∀α^E.T) =
  (* α is fresh, because we open the binder *)
```

```
  let (P; Δ; τ) = tr_type(T) in
  fresh P_s = {p_β | β ∈ Δ} in
  fresh Δ' = {γ_β | β ∈ Δ} in
  (P ⊎ P_s; Δ'; ∀α^E. {β ∈ Δ ↦ γ_β · α?p_β}^* τ)

tr_type(T_1 →_E T_2) =
  let (P_1; Δ_1; τ_1) = tr_type(T_1) in
  let (P_2; Δ_2; τ_2) = tr_type(T_2) in
  let (Δ_3; ε) = tr_effect(E) in
  (P_1 ⊎ P_2; Δ_1 ⊎ Δ_2 ⊎ Δ_3; τ_1 →_ε τ_2)
```

As a specification of `tr_type` function we can prove the following two lemmas, that state its soundness and completeness.

Lemma 1. *If* $\texttt{tr\_type}(T) = (P;\ \Delta';\ \tau)$ *for* T *defined over* Δ*, then* $\Delta, \Delta' \vdash_\rho T \sim \tau$ *for each valuation* ρ*.*

Lemma 2. *If* $\Delta \vdash T \sim \tau$ *then* $\texttt{tr\_type}(T) = (P;\ \Delta';\ \tau')$ *for some* P*,* Δ'*,* τ'*. Moreover, there exists valuation* ρ *and substitution* θ *such that* $\Delta; \cdot \vdash_\rho \theta^*\tau' <: \tau$ *and* $\Delta; \cdot \vdash_\rho \tau <: \theta^*\tau'$*.*

```
subtype(τ_0; τ_0') =
  match (τ_0; τ_0') with
  | (α; α') =>
    if α = α' then (∅; ⊤) else fail
  | (∀α^T. τ; ∀α^T. τ') =>
    let (Ω, φ) = subtype(τ, τ') in
    (Ω; φ)
  | (∀α^E. τ; ∀α^E. τ') =>
    let (Ω, φ) = subtype(τ, τ') in
    ({α ↦ ι}^* Ω; φ ∧ Ω[α])
  | (τ_1 →_ε τ_2; τ_1' →_ε' τ_2') =>
    let (Ω_1, φ_1) = subtype(τ_1', τ_1) in
    let (Ω_2, φ_2) = subtype(τ_2, τ_2') in
    (Ω_1 ∪ Ω_2 ∪ {ε <: ε'}; φ_1 ∧ φ_2)
  | _ => fail
```

To ensure soundness and completeness of this procedure we prove the following two lemmas.

Lemma 3. *If* $\texttt{subtype}(\tau_1;\ \tau_2) = (\Omega;\ \varphi)$ *for types* τ_1*,* τ_2 *defined over* Δ*, then* $\Delta; \Omega \vdash_\rho \tau_1 <: \tau_2$ *for every valuation* ρ *satisfying* φ*.*

Lemma 4. *If* $\Delta; \Omega \vdash_\rho \tau_1 <: \tau_2$*, then* $\texttt{subtype}(\tau_1;\ \tau_2) = (\Omega';\ \varphi)$ *for some* Ω' *and* φ *such that* $\rho \models \varphi$ *and* $\Delta; \Omega \vdash_\rho \Omega'$*.*

```
norm(α <: ε; φ) = {α?φ <: ε}
norm(ι <: ε; φ) = ∅
norm(ε₁ · ε₂ <: ε; φ) = norm(ε₁ <: ε; φ) ∪ norm(ε₂ <: ε; φ)
norm(ε₁?ψ <: ε; φ) = norm(ε₁ <: ε; ψ ∧ φ)

normalize(Ω) = ⋃_{(ε₁<:ε₂)∈Ω} norm(ε₁ <: ε₂; ⊤)

separate(Δ_g; Ω) =
  let Ω_n = normalize(Ω) in
  let Ω_g = {γ?φ <: ε | (γ?φ <: ε) ∈ Ω_n ∧ γ ∈ Δ_g} in
  let Ω_p = {β?φ <: {γ ∈ Δ_g ↦ ι}*ε | (β?φ <: ε) ∈ Ω_n ∧ β ∉ Δ_g} in
  (Ω_g; Ω_p)

inst(∀Δ.[Ω] τ) = (Δ; Ω; τ)

infer(Γ; x) =
  let (∀Δ₁.[Ω₁] τ) = Γ(x) in
  (∅; Δ₁; τ; ι; Ω₁; ⊤)

infer(Γ; λx : T.e) =
  let (P₁; Δ₁; τ₁) = tr_type(T) in
  let (P₂; Δ₂; τ₂; ε; Ω; φ) = infer(Γ, x : τ₁; e) in
  (P₁ ⊎ P₂; Δ₁ ⊎ Δ₂; τ₁ →_ε τ₂; ι; Ω; φ)

infer(Γ; Λα^T.e) =
  let (P₁; Δ₁; τ; ε; Ω₁; φ₁) = infer(Γ; e) in
  (P₁; Δ₁; ∀α^T.τ; ι; Ω₁ ∪ {ε <: ι}; φ₁)

infer(Γ; Λα^E.e) =
  let (P₁; Δ₁; τ; ε; Ω₁; φ₁) = infer(Γ; e) in
  fresh P_s = {p_β | β ∈ Δ₁} in
  fresh Δ₁' = {γ_β | β ∈ Δ₁} in
  let θ = {β ∈ Δ₁ ↦ γ_β · α?p_β} in
  let Ω₁' = θ*(Ω₁ ∪ {ε <: ι}) in
  (P₁ ⊎ P_s; Δ₁'; ∀α^E.θ*τ; ι; {α ↦ ι}*Ω₁'; φ₁ ∧ Ω₁'[α])

infer(Γ; e₁ e₂) =
  let (P₁; Δ₁; τ₁; ε₁; Ω₁; φ₁) = infer(Γ; e₁) in
  match τ₁ with
  | τ_a →_ε τ_v =>
    let (P₂; Δ₂; τ₂; ε₂; Ω₂; φ₂) = infer(Γ; e₂) in
    let (Ω_s, φ_s) = subtype(τ₂; τ_a) in
    (P₁ ⊎ P₂; Δ₁ ⊎ Δ₂; τ_v; ε₁ · ε₂ · ε; Ω₁ ∪ Ω₂ ∪ Ω_s; φ₁ ∧ φ₂ ∧ φ_s)
  | _ => fail
```

```
infer(Γ; e [T]) =
  let (P1; Δ1; τ1; ε1; Ω1; φ1) = infer(Γ; e) in
  match τ1 with
  | ∀α^T.τ =>
    let (P2; Δ2; τ2) = tr_type(T) in
    (P1 ⊎ P2; Δ1 ⊎ Δ2; {α ↦ τ2}*τ; ε1; Ω1; φ1)
  | _ => fail

infer(Γ; e [E]) =
  let (P1; Δ1; τ1; ε1; Ω1; φ1) = infer(Γ; e) in
  match τ1 with
  | ∀α^E.τ =>
    let (Δ2; ε2) = tr_effect(E) in
    (P1; Δ1 ⊎ Δ2; {α ↦ ε2}*τ; ε1; Ω1; φ1)
  | _ => fail

infer(Γ; let x = e1 in e2) =
  let (P1; Δ1; τ1; ε1; Ω1; φ1) = infer(Γ; e1) in
  fresh Δ1' = {β_α | α ∈ Δ1} in
  fresh Δg = {γ_α | α ∈ Δ1} in
  let θ = {α ∈ Δ1 ↦ β_α · γ_α} in
  let (Ωg, Ωp) = separate(Δg; θ*(Ω1 ∪ {ε1 <: ι})) in
  let (P2; Δ2; τ2; ε2; Ω2; φ2) = infer(Γ, x : ∀Δg.[Ωg] θ*τ1; e2) in
  (P1 ⊎ P2; Δ1' ⊎ Δ2; τ2; ε2; Ωp ∪ Ω2; φ1 ∧ φ2)
```