Implementation of type inference for a programming language with algebraic effects

(Inferencja typów dla języka programowania z efektami algebraicznymi)

Dominik Gulczyński

Praca licencjacka

Promotor: dr Filip Sieczkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

8 września 2020

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Contents

Chapter 1

Introduction

Programming languages that enjoy *static type safety* require that each variable and expression is given a *type*. Assigning values to variables of different types usually results in a compile error.

Type inference method was designed to aid the process of *type-checking* and to ease the lives of programmers. It is a process of *deducing* the type of an expression, by the analysis of its structure and the environment in which we carry out the inference process. Many programming languages use it to free programmers from the burden of writing explicit type annotations, which makes code easier to read and modify. For example, in programming language C♯ programmer may use the `var` keyword to omit the type in variable declaration.

A *complete* type inference system can deduce a type for every expression without the need for any type annotations. Most popular of such type systems, called *Hindley-Milner*, is based on the works of J. Roger Hindley[5], Robin Milner, who presented *the* type inference algorithm $\mathbb{W}$[4], and Luis Damas[3]. Their work is a corner stone of type theory for programming languages and the basis for type systems and type inference present in most of the statically typed functional programming languages.

Type systems like System-F or Hindley-Milner's, based on the *lambda calculus* are great mathematical models of describing pure computations and types of pure terms concisely. Yet, a lot of real-life programs heavily use side effects, be it writing to memory, performing I/O actions, or mutating state, and those systems cannot express them. There are different ways of describing side effects in both scientific literature and real world implementations of programming languages. Some languages do not restrict side effects at all, like OCaml, while others have their unique ways of expressing side effects, like monadic actions in Haskell. Finally there are so called type-and-effect systems.

The particular system presented by Biernacki et al[1] uses *algebraic effects* to describe programming with computational effects. An effect is defined by a set of available operations (defined by its *signature*) and a handler that provides its implementation. Operators are meaningless on their own, acting much like ordinary functions calling the handler's body, while the handler itself defines the way operation calls execute.

One could think of effects as a generalization of exceptions: calling an operator corresponds to throwing an exception, and enclosing expression by a handler, corresponds to *try* {*...*} *catch* {*...*} construct known in some form in many mainstream languages. But, in most of those languages exceptions are not *resumable*, meaning that once we *leave* the inner expression of a handler, by calling one of its operators, we cannot continue the computation of handled expression.

We transcribe the calculus given by Biernacki et al to the world of ML programming by presenting modified type-and-effect system and corresponding calculus, together with type inference algorithm for it.

Chapter 2

Calculus

The calculus we present is a subset of work by Biernacki et al[1]. It is an extension of standard *call-by-value lambda calculus with let* by effect handlers and operators. We adjusted it to match the style of *ML-the-calculus* and *ML-the-type-system*[2].

$$\begin{array}{rll}
\mathbf{var} \ni x, \dots & & \text{(term variables)} \\
\mathbf{qvar} \ni \alpha, \dots & & \text{(quantified variables)} \\
\mathbf{tvar} \ni t, \dots & & \text{(type variables)} \\
\mathbf{evar} \ni \epsilon, \dots & & \text{(effect variables)} \\
\mathbf{ivar} \ni a, \dots & & \text{(instance variables)} \\
\mathbf{type} \ni \tau ::= & \alpha \mid t \mid \mathbf{Unit} \mid \mathbf{Int} \mid \tau \to_{\varepsilon} \tau \mid \forall (a : \sigma).\, \tau & \text{(types)} \\
\mathbf{scheme} \ni \pi ::= & \forall \alpha.\, \pi \mid \tau & \text{(type schemes)} \\
\mathbf{effect} \ni \varepsilon ::= & \alpha \mid \epsilon \mid a \mid \varepsilon \circ \varepsilon & \text{(effects)} \\
\mathbf{signature} \ni \sigma ::= & \mathbf{Error} \mid \mathbf{State}\ \tau & \text{(signatures)} \\
\mathbf{expr} \ni e ::= & x \mid () \mid n \mid \lambda x.\, e \mid \mathbf{fun}\ f\ x.\, e \mid e\ e \mid \mathbf{let}\ x = e\ \ \mathbf{in}\ \ e & \text{(terms)} \\
& \mid \lambda (a : \sigma).\, e \mid e\ a \mid op_a\ e \mid \mathbf{handle}_a\ e\ \{h; \mathbf{return}\ x.e\} & \\
\mathbf{operator} \ni op ::= & \mathbf{Raise} \mid \mathbf{Get} \mid \mathbf{Put} & \text{(operators)} \\
\mathbf{handler} \ni h ::= & [(\mathbf{Raise}, x, k.e)] \mid [(\mathbf{Get}, x, k.e); (\mathbf{Put}, x, k.e)] & \text{(handlers)}
\end{array}$$

Terms (or expressions) are given by:

- variables, bound by abstractions, let-expressions, handlers, or environment Γ,
- constants: $()$ and $n \in \mathbb{C}$,
- abstractions: anonymous functions $\lambda x.\, e$ with argument x and body e and recursive functions denoted $\mathbf{fun}\ f\ x.\, e$,
- instance abstraction: $\lambda(a : \sigma).\, e$, that lets programmers write code unspecific to certain effect *instance*, but capable of working with any instance of specified signature,

- applications: $e_1\,e_2$ and $e\,a$, for applying arguments to respective abstractions,
- let-construct: **let** $x = e_1$ **in** e_2, which first evaluates body of e_1, and bounds its value to variable x in expression e_2,
- operation calls: $op_a\;e$ calling the op operator handler of instance a with value of e,
- handlers: $\mathbf{handle}_a\,e\{h; \mathbf{return}\ x.e_r\}$ of instance a, which provides meaning to operators: calling $op_a\,e_x$, executes body e_{op} of a construct $(op, x, k.\,e_{op})$ defined in handler h.

Instead of allowing effects of arbitrary signatures, we limited them to instances of either **Error** or **State** τ. Arbitrary signatures could be dealt with in similar fashion as *ADTs* (algebraic data types) and are *orthogonal* to type inference.

For the *complete type inference* we need to limit the system to *prenex* polymorh-pism and thus we differentiate between *monomorphic* types and *polymorphic* type schemes (which are types proceeded by α variables). Accordingly we have dropped the type-lambdas (Λ) from calculus in favor of the polymorphic let-construct.

The semantics of calculus strictly follows the rules defined in Biernacki et al. For formal reduction rules and additional insight, see “Binders by Day, Labels by Night”[1].

Chapter 3

Type system

We have tweaked the type-and-effect system constructed by Biernacki et al[1] to match the restrictions required by classical methods of type inference. Thus, we introduce *unification variables* for both types and effects (written t and ϵ, respectively). We use these variables to denote "yet to be determined" types. During the type inference we will *generate* these unification variables, and deduce rigid types for them, as described in Chapter 4.

Judgement $\Gamma, \Theta \vdash e\ : \tau/\varepsilon$ states that in environments Γ (assigning types to variables), and Θ (assigning signatures to instances), term e *inhabits* syntactical type and effect τ/ε (which means that computing e would yield a value of type τ and possibly cause effect ε).

Typing terms:

$$\frac{}{\Gamma; \Theta \vdash ()\ : \mathbf{Unit}/\iota} \qquad \frac{}{\Gamma; \Theta \vdash n\ : \mathbf{Int}/\iota}$$

There are two *base types* **Unit** and **Int** for constants.

$$\frac{(x : \pi) \in \Gamma \qquad \pi[\vec{\tau_\alpha}/\vec{\alpha}] = \tau}{\Gamma; \Theta \vdash x\ : \tau/\iota}$$

Judgement for variables follows the usual let-polymorphism typing, where variables bound by let clauses are generalized and need to be instantiated. Variables do not cause effects as only the value is assigned to them, while the effects caused by their computation (if any occur) are bound to the term which introduced that variable.

$$\frac{\Gamma, (x : \tau_1); \Theta \vdash e\ : \tau_2/\varepsilon}{\Gamma; \Theta \vdash \lambda x.\, e\ : \tau_1 \rightarrow_\varepsilon \tau_2/\iota} \qquad \frac{\Gamma, (f : \tau_1 \rightarrow_\varepsilon t_2), (x : \tau_1); \Theta \vdash e\ : \tau_2/\varepsilon}{\Gamma; \Theta \vdash \mathbf{fun}\, f\, x.\, e\ : \tau_1 \rightarrow_\varepsilon \tau_2/\iota}$$

The *type constructor* $\rightarrow$ is used to type abstractions, where type $\tau_1 \rightarrow_\varepsilon \tau_2$ is given to functions that when applied with input of type τ_1, computing the function's body e produces some output of type τ_2, possible causing effect ε

$$\frac{\Gamma;\Theta\vdash e_1\ :\tau_2\rightarrow_\varepsilon\tau/\varepsilon_1\qquad\Gamma;\Theta\vdash e_2\ :\tau_2/\varepsilon_2}{\Gamma;\Theta\vdash e_1\,e_2\ :\tau/\varepsilon_1\circ\varepsilon\circ\varepsilon_2}$$

Accordingly, effect of application combines effects of: computing the left-hand side term, effect that "hangs" on it's arrow type, and the effect of computing the right-hand side term.

$$\frac{\Gamma;\Theta\vdash e_1\ :\tau_1/\iota\qquad\mathrm{gen}(\Gamma,\tau_1)=\pi\qquad\Gamma,(x:\pi);\Theta\vdash e_2\ :\tau/\varepsilon}{\Gamma;\Theta\vdash\mathbf{let}\ x=e_1\ \mathbf{in}\ e_2\ :\tau/\varepsilon}$$

$$\frac{\Gamma;\Theta\vdash e_1\ :\tau_1/\varepsilon_1\qquad\varepsilon_1\neq\iota\qquad\Gamma,(x:\tau_1);\Theta\vdash e_2\ :\tau/\varepsilon}{\Gamma;\Theta\vdash\mathbf{let}\ x=e_1\ \mathbf{in}\ e_2\ :\tau/\varepsilon}$$

As usually in let-polymorphism schemes, we *generalize* the type derived for e_1 before we add it to the environment in which we derive type for e_2. This is achieved by the function $\mathrm{gen}(\Gamma,\tau)=\forall\vec{\alpha}.\ \tau\ \textit{where}\ \vec{\alpha}=\mathrm{freevars}(\tau)\setminus\mathrm{freevars}(\Gamma)$. We restrict generalization to only *pure* terms, i.e. such that their computation would cause no effects.

$$\frac{\Gamma;\Theta,(a:\sigma)\vdash e\ :\tau/\iota}{\Gamma;\Theta\vdash\lambda(a:\sigma)..e\ :\forall(a:\sigma).\tau/\iota}\qquad\frac{\Gamma;\Theta\vdash e\ :\forall(a:\sigma).\tau/\iota\qquad(b:\sigma)\in\Theta}{\Gamma;\Theta\vdash e\,b\ :\tau[b/a]/\iota}$$

For handling operators of distinct occurrences of the same signature, i.e many cells of memory via **State** τ, we differentiate them by assigning them different instances. *Instance abstractions* allow us to write code that works not only with one instance of particular signature, but with any that is supplied using *instance application* of instances bound by handlers or other instance lambdas.

$$\frac{\Theta\vdash op_a:\tau_e\Rightarrow\tau\qquad\Gamma;\Theta\vdash e\ :\tau_e/\varepsilon}{\Gamma;\Theta\vdash op_a\,e\ :\tau/a\circ\varepsilon}$$

Operators of type $\tau_1\Rightarrow\tau_2$ expect arguments of type τ_1 and return value of type τ_2. Type of e must match the operator type. Effect of computing e is extended by $\{a\}$.

$$\frac{\Gamma;\Theta\vdash h\ :\sigma\rhd\tau/\varepsilon\qquad\Gamma;\Theta,(a:\sigma)\vdash e\ :\tau'/a\circ\varepsilon\qquad\Gamma,(x:\tau');\Theta\vdash e_r\ :\tau/\varepsilon}{\Gamma;\Theta\vdash\mathbf{handle}_a\,e\,\{h;\mathbf{return}\ x.e_r\}\ :\tau/\varepsilon}$$

For type-checking handler body, there's a separate relation $\vdash\cdot:\cdot\rhd\cdot$, explained below. Types of all *exectution paths* of the handler must match, whether the value returns from within handler body or via return expression e_r (where value of evaluating e gets bound to x).

Finally, we allow every type and effect to *grow* as needed:

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

There are two clauses for typing handlers, as we only have kinds of signatures. It could easily extended for other signatures but that's not important to this work.

$$\frac{\Gamma, (x : \textbf{Unit}), (k : \tau' \to_\varepsilon \tau); \Theta \vdash e \ : \tau/\varepsilon}{\Gamma; \Theta \vdash [(\textbf{Raise}, x, k.e)] \ : \textbf{Error} \triangleright \tau/\varepsilon}$$

$$\frac{\begin{array}{c}\Gamma, (x : \textbf{Unit}), (k : \tau' \to_\varepsilon \tau); \Theta \vdash e' \ : \tau/\varepsilon \\ \Gamma, (x : \tau'), (k : \textbf{Unit} \to_\varepsilon \tau'); \Theta \vdash e'' \ : \tau/\varepsilon\end{array}}{\Gamma; \Theta \vdash [(\textbf{Get}, x, k.e'); (\textbf{Put}, x, k.e'')] \ : \textbf{State}\ \tau' \triangleright \tau/\varepsilon}$$

Supplying the *continuation* k with some value v continues evaluation of e, with v substituted in place of operation call, thus in the body of operator handler of type $\tau_1 \Rightarrow \tau_2$, k is given type $\tau_2 \to_\varepsilon \tau$ and x is given τ_2. In a sense k acts just like an ordinary function, and programmer may use it in many different ways or not even use it all, returning a value straight from the handler code. which is the final value returned by handler. However they choose to do so, the type-and-effect of handlers expression must match the type of the whole handled expression.

$$\frac{(a : \textbf{Error}) \in \Theta}{\Theta \vdash \textbf{Raise}_a : \textbf{Unit} \Rightarrow \tau} \qquad \frac{(a : \textbf{State}\ \tau) \in \Theta}{\Theta \vdash \textbf{Put}_a : \textbf{Unit} \Rightarrow \tau} \qquad \frac{(a : \textbf{State}\ \tau) \in \Theta}{\Theta \vdash \textbf{Get}_a : \tau \Rightarrow \textbf{Unit}}$$

Relation $\Theta \vdash op_a : \tau_1 \Rightarrow \tau_2$ simply finds the signature σ bound to instance a in environment Θ and returns the operator's type.

3.1 Subtyping

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1' <: \tau_1 \quad \varepsilon <: \varepsilon' \quad \tau_2 <: \tau_2'}{\tau_1 \to_\varepsilon \tau_2 <: \tau_1' \to_{\varepsilon'} \tau_2'} \qquad \frac{}{\varepsilon <: \varepsilon} \qquad \frac{\varepsilon <: \varepsilon'}{\varepsilon <: \varepsilon' \circ \varepsilon''}$$

The subtyping rule we propose is *structural*, meaning that only types of *matching shape* are related. However, while leaves containing types must be equal, we allow effects to differ as long as they are related. Notice that the $\to$ is contravariant to subtyping relation (in the premise of rule for $\to$, the relation on τ_1 and τ_1' is reversed).

The subtyping plays a vital role in usability of the calculus. Consider a term f, a function that does some calculatio but allowed the function to fail, i.e.

$$\emptyset, (e : Error) \vdash f \ : (\textbf{Int} \to_e \textbf{Int}) \to_e \textbf{Int}/\iota$$

but nothing stops us from applying some pure function in this place. On the other hand, it would be undesirable if we could supply a term expecteing a pure function and with an effectful one. Clearly this property gives us flexibility, while keeping effects under control.

3.2 Parametricity

Our type system maintains predicative prenex polymorphism of ML, extended with universal quantification over effects because original paper maintains it.

3.3 Principal type

In ML type system, the *principal type* property states that there exists a *most general* type for any correct program[3]. A type scheme π is called *principal* if any other type that could be given to e is an *instantiaton* of it. For example, consider $e = \lambda x.\, x$. There's a few types that could be given to it: **Int** $\to$ **Int**, but clearly any correct type we would think of would not be more general than $\forall\alpha.\alpha \to \alpha$.

Definition 3.3.1. Principal Type
In our type system, π is a principal type of e in environments Γ, Θ iff

$$\forall\vec{\tau}.\, \Gamma;\Theta \vdash e\ : \pi[\vec{\tau}/\vec{\alpha}]/\iota\ \wedge\ \forall\pi'.\, \exists\vec{\tau'}.\, \Gamma;\Theta \vdash e\ : \pi'[\vec{\tau'}/\vec{\alpha'}]/\iota\ \wedge\ \pi[\vec{\tau}/\vec{\alpha}] <: \pi'[\vec{\tau'}/\vec{\alpha'}]$$

For details about subtyping and principal type, see future work.

For now we will give an example of how to approximate principal type. Consider two type schemes that could be assigned to a term *compose* $= \lambda f.\, \lambda g.\, \lambda x.\, f(g\ x)$, representing function composition:

$$\pi_1 ::= \forall\alpha,\beta.\, (\tau_b \to_\alpha \tau_c) \to (\tau_a \to_\beta \tau_b) \to \tau_a \to_{\alpha\circ\beta} \tau_c$$
$$\pi_2 ::= \forall\gamma.\, (\tau_b \to_\gamma \tau_c) \to (\tau_a \to_\gamma \tau_b) \to \tau_a \to_\gamma \tau_c$$

At first glance, it may look like π_1 is the "correct" type for *compose*, as it seems more natural, i.e. given functions f causing effect ε_f, and g causing ε_f, *compose* f g is a function that would apply them both, so it clearly must be causing effect $\varepsilon_f \circ \varepsilon_g$.

While this reasoning is sound, we are interested in deriving the most concise type possible. Let's see if there exists ε_h such that π_1 instantiated with arbitrary effects ε_f, ε_g subtypes π_2 instantiated with ε_h:

$$\pi_1[\varepsilon_f,\varepsilon_g/\alpha,\beta] <: \pi_2[\varepsilon_h/\gamma]$$
$$\Longleftrightarrow$$
$$(\tau_b \to_{\varepsilon_f} \tau_c) \to (\tau_a \to_{\varepsilon_g} \tau_b) \to \tau_a \to_{\varepsilon_f\circ\varepsilon_g} \tau_c <: (\tau_b \to_{\varepsilon_h} \tau_c) \to (\tau_a \to_{\varepsilon_h} \tau_b) \to \tau_a \to_{\varepsilon_h} \tau_c$$
$$\Longleftrightarrow$$
$$\varepsilon_h <: \varepsilon_f\ \wedge\ \varepsilon_h <: \varepsilon_g\ \wedge\ \varepsilon_f \circ \varepsilon_g <: \varepsilon_h$$

Clearly, $\pi_1[\varepsilon_f,\varepsilon_g/\alpha,\beta] <: \pi_2[\varepsilon_h/\gamma]$ does not hold for ε_f and ε_g other than ι, thus π_1 cannot be a principal type of e. On the other hand, if we were to check if for arbitrary ε_h there exist ε_f and ε_g such that $\pi_2[\varepsilon_h/\gamma] <: \pi_1[\varepsilon_f, \varepsilon_g/\alpha, \beta]$, we would need to find witnesses for such formula:

$$\varepsilon_f <: \varepsilon_h \ \wedge \ \varepsilon_g <: \varepsilon_h \ \wedge \ \varepsilon_h <: \varepsilon_f \circ \varepsilon_g$$

Clearly if we choose $\varepsilon_f = \varepsilon_g = \varepsilon_h$, it is satisfied, which means that π_2 is indeed more general π_1. We designed our inference algorithm with this intuition in mind and π_2 is the desired result that our implementation actually infers.

Chapter 4

Inference algorithm

The algorithm we present loosely follows original algorithm $\mathbb{W}$, executing in two distinct phases:

1. Constraint gathering: the algorithm traverses the structure of given expression, generating sub-expressions' types, effects and constraints,
2. Constraint solving: the algorithm builds a substitution that resolves gathered constraints, in respect to the laws of type checking.

In practice, these phases are often interleaved, as even in pure *let-polymorphism* type interference when handling expression **let** $x = e_1$ **in** e_2 we need to solve constraints regarding the inferred type of e_1, so we can *generalize* it before adding it to the environment for inferring e_2.

$$\begin{aligned}\textbf{constraints} \ni C &::= \emptyset \mid \{\tau <: \tau\} \mid \{\varepsilon <: \varepsilon\} \mid C \cup C \\ \textbf{substitution} \ni S &::= id \mid [t \mapsto \tau] \mid [\epsilon \mapsto \varepsilon] \mid S \circ S\end{aligned}$$

Constraints $\{x <: y\}$ are a way of describing necesarry conditions that need to be satisfied for the expresion and inferred type to type-check. A substitution is a mapping from type and effect *unification variables* to inferred types or effects, respectively. We will write $S[t \mapsto \tau]$ for $[t \mapsto \tau] \circ S$. Substitution id is simply an identity function.

4.1 Remarks on effect unification

With combining effect *unification variables* and constraint solving, a problem arises. Consider constraint $a <: \epsilon_1 \circ \epsilon_2$, for some instance variable a and some effect unification variables ϵ_1, ϵ_2. To resolve such constraint we have a few viable options:

1. Expand ϵ_1 with a.
2. Expand ϵ_2 with a.
3. Expand both ϵ_1 and ϵ_2 with a.

But how would we choose one over the other? Maybe one of those makes the program ill-typed, while the others do not? What if there's more than two unification variables? Clearly such constraints are undesirable.

To tackle this problem, in our algorithm we permit no more than one effect *unification variable* or *quantified variable* in effects. Thus the effects are defined differently than in Chapter 1:

$$\begin{aligned} \textbf{effects} \ni \varepsilon &::= I \mid I * \alpha \mid I * \epsilon && \text{(effects)} \\ I &::= \iota \mid \{a\} \mid I \cup I \mid I \setminus I && \text{(sets of instances)} \end{aligned}$$

So an effect is either just a finite set of instances, or a union of one with either effect *unification variable* or *generalized* effect variable. If we think about effects as sets, then the subtyping relation of effects simply boils down to set inclusion. Notice how every effect defined by the new grammar is expressible in the previous one as well.

4.2 Generating constraints

As in algorithm $\mathbb{W}$, to infer the type for a given term, we build it by working *bottom-up* from the leaves through the whole expression tree. To this end, we have defined judgement $\Gamma; \Theta \vdash_{\mathbf{Gen}} e : \tau/\varepsilon \rightsquigarrow C; S$ that states in the the premise what conditions need to be satisfied for deducing type, and under what constraints C and substitution S shall we interpret it.

We chose to present $\vdash_{\mathbf{Gen}}$ typing rules as close as possible to the practical inference algorithm. Hence, we heavily use *unification variables* (denoted t for types and ϵ for effects), and the judgement should be thought of as a semi-formal description of type and constraints generating process (working bottom-up) rather than a type checker ($\vdash$ working top-down).

It is important that we "return" not only constraints, but also a substitution, because some constraints may have been already resolved in one sub-tree of the term, effectively changing the environment, and we need to take it into account while inferring the other sub-trees. We abstract solving constraints C in environment Γ (under substitution S) to a high-level functions *solve* and *solve'* which return a reduced set of constraints C' and a new substitution S'.

Definition 4.2.1. Syntactic type soundness

$$\begin{aligned}&\Gamma;\Theta \vdash_{\mathbf{Gen}} e \ : \tau/\varepsilon \rightsquigarrow C; S \implies \\ &\qquad \mathrm{solve}(\Gamma, \tau/\varepsilon, C, S) = \emptyset; S' \implies \\ &\qquad\qquad S'\Gamma; S'\Theta \vdash e \ : S'\tau/S'\varepsilon\end{aligned}$$

We would say that an inference algorithm enjoys *syntactic type soundness* if the type generated by it (after we have solved all the constraints and built the substitution) for the given term checks by the syntactic rules we have defined in chapter 3. Proof of our algorithm is left for future work.

$$\frac{}{\Gamma;\Theta \vdash_{\mathbf{Gen}} () \ : \mathbf{Unit}/\iota \rightsquigarrow \emptyset; id} \qquad \frac{}{\Gamma;\Theta \vdash_{\mathbf{Gen}} n \ : \mathbf{Int}/\iota \rightsquigarrow \emptyset; id}$$

$$\frac{(x : \pi) \in \Gamma \qquad \mathrm{inst}(\pi) = \tau}{\Gamma;\Theta \vdash_{\mathbf{Gen}} x \ : \tau/\iota \rightsquigarrow \emptyset; id}$$

As environment Γ contains type schemes rather than types, before we return the type of x, we switch out every variable quantified in π to a fresh unification variable via function *inst*.

$$\frac{\Gamma, (x : t);\Theta \vdash_{\mathbf{Gen}} e \ : \tau/\varepsilon \rightsquigarrow C; S \qquad \mathrm{fresh}(t)}{\Gamma;\Theta \vdash_{\mathbf{Gen}} \lambda x.\, e \ : t \to_{\varepsilon} \tau/\iota \rightsquigarrow C; S}$$

$$\frac{\begin{array}{c}\mathrm{fresh}(t_1) \qquad \mathrm{fresh}(\epsilon) \qquad \mathrm{fresh}(t_2) \\ \Gamma, (f : t_1 \to_{\epsilon} t_2), (x : t_1);\Theta \vdash_{\mathbf{Gen}} e \ : \tau/\varepsilon \rightsquigarrow C; S\end{array}}{\Gamma;\Theta \vdash_{\mathbf{Gen}} \mathbf{fun}\ fx.e \ : t_1 \to_{\epsilon} t_2/\iota \rightsquigarrow C \cup \{\varepsilon <: \epsilon, \tau <: t_2\}; S}$$

For recursive functions, we must ensure that the actual type-and-effect of function's body e subtypes one we have generated for f.

$$\frac{\begin{array}{c}\Gamma;\Theta \vdash_{\mathbf{Gen}} e_1 \ : \tau_1/\varepsilon_1 \rightsquigarrow C_1; S_1 \qquad S_1\Gamma; S_1\Theta \vdash_{\mathbf{Gen}} e_2 \ : \tau_2/\varepsilon_2 \rightsquigarrow C_2; S_2 \\ \mathrm{fresh}(t) \qquad \mathrm{fresh}(\epsilon) \qquad C = C_1 \cup C_2 \cup \{\varepsilon_1 <: \epsilon, \varepsilon_2 <: \epsilon, \tau_1 <: \tau_2 \to_{\epsilon} t\}\end{array}}{\Gamma;\Theta \vdash_{\mathbf{Gen}} e_1\, e_2 \ : t/\epsilon \rightsquigarrow C; S_2 \circ S_1}$$

As mentioned earlier, we use the generated substitution S_1 to update the environment for handling e_2. Additional constraints ensure that the fresh effect variable we return is sybtyped by the effects of computing e_1, e_2, and the effect hung on the $\to$ type of e_1.

$$\frac{\begin{array}{c}\Gamma;\Theta \vdash_{\mathbf{Gen}} e_1 \ : \tau_1/\varepsilon_1 \rightsquigarrow C_1; S_1 \qquad \mathrm{solve}(\Gamma, \tau_1/\varepsilon_1, C_1, S_1) = C; S \\ S\varepsilon_1 = \iota \qquad \mathrm{gen}(S\Gamma, S\tau_1) = \pi \qquad S\Gamma, (x : \pi); S\Theta \vdash_{\mathbf{Gen}} e_2 \ : \tau_2/\varepsilon_2 \rightsquigarrow C_2; S_2\end{array}}{\Gamma;\Theta \vdash_{\mathbf{Gen}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \ : \tau_2/\varepsilon_2 \rightsquigarrow C \cup C_2; S_2 \circ S}$$

$$\frac{\begin{array}{c}\Gamma;\Theta \vdash_{\mathbf{Gen}} e_1 \ : \tau_1/\varepsilon_1 \rightsquigarrow C_1; S_1 \qquad \mathrm{solve}(\Gamma, \tau_1/\varepsilon_1, C_1, S_1) = C; S \\ S\varepsilon_1 \neq \iota \qquad S\tau_1 = \tau \qquad S\Gamma, (x : \tau); S\Theta \vdash_{\mathbf{Gen}} e_2 \ : \tau_2/\varepsilon_2 \rightsquigarrow C_2; S_2 \qquad \mathrm{fresh}(\epsilon)\end{array}}{\Gamma;\Theta \vdash_{\mathbf{Gen}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \ : \tau_2/\epsilon \rightsquigarrow C \cup C_2 \cup \{\varepsilon_1 <: \epsilon, \varepsilon_2 <: \epsilon\}; S_2 \circ S}$$

Before we can handle e_2, we need to check if type generatod for e_1 can be generalized. To this end, we deploy the *solve* function, and if the substitutin returned ensured us computing of e_1 would cause no effects, we can safely generalize it via *gen* function, otherwise we take its effect into account, adding constraints regarding fresh variable ϵ. Again, S is applied to environments for handling e_2.

$$\frac{\Gamma; \Theta, (a : \sigma) \vdash_{\mathbf{Gen}} e \ : \tau/\varepsilon \rightsquigarrow C; S}{\Gamma; \Theta \vdash_{\mathbf{Gen}} \lambda(a : \sigma).\, e \ : \forall(a : \sigma).\tau/\iota \rightsquigarrow C \cup \{\varepsilon <: \iota\}; S}$$

$$\frac{\begin{array}{c}\Gamma; \Theta \vdash_{\mathbf{Gen}} e \ : \tau'/\varepsilon \rightsquigarrow C'; S' \qquad \text{solve'}(\Gamma, C', S') = C; S \\ S\tau' = \forall(a : \sigma).\tau \qquad S\varepsilon = \iota \qquad (b : \sigma) \in S\Theta\end{array}}{\Gamma; \Theta \vdash_{\mathbf{Gen}} e\, b \ : \tau[b/a]/\iota \rightsquigarrow C; S}$$

For instance abstraction, we have to make sure that the expression underneath it is pure. With instance application, we need to *know* that the type for e is quantified by some instance a, so we use the weaker *solve'* function to find out its type, before substituting a for b in it.

$$\frac{\Theta \vdash op_a : \tau_1 \Rightarrow \tau_2 \qquad \Gamma; \Theta \vdash_{\mathbf{Gen}} e \ : \tau_e/\varepsilon \rightsquigarrow C; S}{\Gamma; \Theta \vdash_{\mathbf{Gen}} op_a\, e \ : \tau_2/a * \varepsilon \rightsquigarrow C \cup \{\tau_e <: \tau_1\}; S}$$

Operators resemble functions, so we make sure that the type of e subtypes the type for arguments of op_a. Effect returned is the consequence of computing e combined with $\{a\}$.

$$\frac{\begin{array}{c}\Gamma; \Theta \vdash_{\mathbf{Gen}} h \ : \sigma \triangleright t/\epsilon \rightsquigarrow C_h; S_h \\ S_h\Gamma; S_h\Theta, (a : S_h\sigma) \vdash_{\mathbf{Gen}} e \ : \tau/\varepsilon \rightsquigarrow C_e; S_e \\ S_eS_h\Gamma, (x : \tau); S_eS_h\Theta \vdash_{\mathbf{Gen}} e_r \ : \tau_r/\varepsilon_r \rightsquigarrow C_r; S_r \\ \text{fresh}(t) \qquad \text{fresh}(\epsilon) \qquad C = C_h \cup C_e \cup C_r \cup \{\varepsilon <: a * \epsilon, \tau_r <: t, \varepsilon_r <: \epsilon\}\end{array}}{\Gamma; \Theta \vdash_{\mathbf{Gen}} \mathbf{handle}_a\ e\ \{h; \mathbf{return}\ x.\ e_r\} \ : t/\epsilon \rightsquigarrow C; S_r \circ S_e \circ S_h}$$

Unsurprisingly, this rule looks rather complicated.

$$\frac{\Gamma, (x : \mathbf{Unit}), (k : t \to_\varepsilon \tau); \Theta \vdash_{\mathbf{Gen}} e \ : \tau'/\varepsilon' \rightsquigarrow C; S \qquad \text{fresh}(t)}{\Gamma; \Theta \vdash_{\mathbf{Gen}} [(\mathbf{Raise}, x, k.\ e')] \ : \mathbf{Error} \triangleright \tau/\varepsilon \rightsquigarrow C \cup \{\tau' <: \tau, \varepsilon' <: \varepsilon\}; S}$$

$$\frac{\begin{array}{c}\Gamma, (x : \mathbf{Unit}), (k : t \to_\varepsilon \tau); \Theta \vdash_{\mathbf{Gen}} e' \ : \tau'/\varepsilon' \rightsquigarrow C'; S' \\ S'\Gamma, (x : t), (k : \mathbf{Unit} \to_\varepsilon \tau); S'\Theta \vdash_{\mathbf{Gen}} e'' \ : \tau''/\varepsilon'' \rightsquigarrow C''; S'' \\ C = C' \cup C'' \cup \{\tau' <: \tau, \varepsilon' <: \varepsilon, \tau'' <: \tau, \varepsilon'' <: \varepsilon\} \qquad S = S'' \circ S' \qquad \text{fresh}(t)\end{array}}{\Gamma; \Theta \vdash_{\mathbf{Gen}} [(\mathbf{Get}, x, k.e'); (\mathbf{Put}, x, k.e'')] \ : \mathbf{State}\ t \triangleright \tau/\varepsilon \rightsquigarrow C; S}$$

4.3 Solving constraints

The constraint solving algorithm we present is divided in two sub-procedures:

1. `solve_simple_contraints` C S
   deals with both type- and effect-constraintsm behaving, much like the ordinary HM algorithm. As the rules for type subtyping are somewhat trivial, it can solve them efficiently and environment-agnostically. As we explain in following subsection, there are some effect constraints that are *non-trivial* and cannot be resolved so easily, so they are dealt with by the second procedure:

2. `solve_contraints_within` Γ (τ, ε) C S
   deals with effect-constraints regarding effect unification variables (ϵ) occurring in Γ, τ and ε. The interesting constrains are of form $\{\epsilon_1 <: I * \epsilon_2\}$, as there are many substations that could satisfy one such constraint, but it is not obvious which one is *best* regarding bigger picture. In following subsections we argue how our approach approximates the most general type. For formal methods, see future work.

4.3.1 Simple constraints

We call constraints solved by the first sub-procedure *simple* as the substitution they induce is *minimal*, meaning it is unambiguous that their premises *must* hold for the whole to be correct. Constrains that are *interesting* are *irreducible* by `solve_simple`. Other constraints are either solved or reduced to the *interesting* form. Notice that this procedure resolves all constraints that regard types (which are trivial) and we can use the substitution returned by it to find out *shapes* of types (and effects, to some extent) without *worrying* whether we would break some constraints by resolving them too quickly.

```
let expand $\epsilon$ $I$ $S$=
  if $I = \emptyset$ then $S$
  else $S[\epsilon \mapsto I * \epsilon']$ where $\text{fresh}(\epsilon')$

let solve_simple $C$ $S$ =
  match $C$ with
  | $\emptyset$          $\rightarrow$ $\emptyset; S$
  | $\{\tau_1 <: \tau_2\} \cup C$ $\rightarrow$
    match $S[\tau_1], S[\tau_2]$ with
    | $\tau_1'$, $\tau_2'$ when $\tau_1' = \tau_2'$ $\rightarrow$
      solve_simple $C$ $S$
    | $t$, $\tau$
    | $\tau$, $t$ $\rightarrow$
      solve_simple $C$ $S[t \mapsto \tau]$
    | $\tau_1' \rightarrow_{\varepsilon_1} \tau_1'', \tau_2' \rightarrow_{\varepsilon_2} \tau_2''$ $\rightarrow$
      solve_simple $\{\tau_1' <: \tau_2', \varepsilon_1 <: \varepsilon_2, \tau_1'' <: \tau_2''\} \cup C$ $S$
  | $\{\varepsilon_1 <: \varepsilon_2\} \cup C$ $\rightarrow$
    match $S[\varepsilon_1], S[\varepsilon_2]$ with
    | $\iota$, _ $\rightarrow$ solve_simple $C$ $S$
    | $I_1$, $I_2$
| $I_1$ , $I_2 * \epsilon$ $\to$
    solve_simple $C$ (expand $(I_1 \setminus I_2)$ $\epsilon$ $S$)
| $I_1 * \epsilon_1$ , $I_2 * \epsilon_2$ $\to$
    if $\epsilon_1 = \epsilon_2$ then solve_simple $C$ (expand $(I_1 \setminus I_2)$ $\epsilon_2$ $S$)
    else let $C'; S'$ = solve_simple $C$ (expand $(I_1 \setminus I_2)$ $\epsilon_2$ $S$)
          in $\{\epsilon_1 <: I_2 * \epsilon_2\} \cup C'; S'$
| $I_1 * \epsilon <: I_2$ when $I_1 \subseteq I_2 \to$
    let $C'; S'$ = solve_simple $C$ $S$
      in $\{\epsilon <: I_2\} \cup C'; S'$
```

We can now define function solve' from $\vdash_{\mathbf{Gen}}$ relation:

$$\text{solve'}(C, S) = \texttt{solve\_simple}\, C\, S$$

4.3.2 Interesting constraints

What makes constraints like $(\epsilon_1 <: I_2 * \epsilon_2)$ interesting is that there are many plausible substitutions that satisfy it. For every set of instances $I_1 \subseteq I_2$, substitution $[\epsilon_1 \mapsto I_1]$ or $[\epsilon_1 \mapsto I_1 * \epsilon_2]$ obviously resolves the constraint, but clearly some substitutions are better than others.

As discussed in previous chapter, $\to$ type constructor is *contravariant* to subtyping relation, which plays a great role in how we treat effect unification variables. During computation of `solve_constraints_within` Γ τ/ε we keep information about *variance* of effect unification variables as a function V.

$$\textbf{variance} \ni v ::= \oplus \mid \odot \mid \ominus \mid \times$$

If variable ϵ appears in Γ, τ, and ε only in *covariant* (*positive*) positions, then $V(\epsilon) = \oplus$. If it appears only *contravariantly* (*negatively*) then $V(\epsilon) = \ominus$. If it appears *invariantly* (*both* positively and negatively), then $V(\epsilon) = \odot$. Finally, if ε doesn't appear in type or environment at all, then $V(\epsilon) = \times$.

Because of the way we defined subtyping relation, we can *shrink* any covariant effect and *expand* any contravariant effect. Consider type τ with some covariant effect $\varepsilon_\oplus$. Clearly, for any $\varepsilon'_\oplus <: \varepsilon_\oplus$ we have $\tau[\varepsilon'_\oplus/\varepsilon_\oplus] <: \tau$. Analogously, for any contravariant effect $\varepsilon_\ominus$ and any effect $\varepsilon'_\ominus$ such that $\varepsilon_\ominus <: \varepsilon'_\ominus$ we have $\tau[\varepsilon'_\ominus/\varepsilon_\ominus] <: \tau$.

With this intuition in mind, we can confidently resolve constraints $\{\epsilon_1 <: I_2 * \epsilon_2\}$ if either $V(\epsilon_1) = \oplus$ or $V(\epsilon_2) = \ominus$ by assignment $[\epsilon_1 \mapsto I_2 * \epsilon_2]$.

It is important to include the non-appearing variables in our algorithm as well, as there are many unification variables that are generated along the way that do not appear in the examined type nor environment explicitly, but often do form *chains* of subtyping constraints like so (instances ommited for better visibility):

$$\epsilon_\ominus <: \epsilon_1 <: \ldots <: \epsilon_n <: \epsilon_\oplus \text{ where } V(\epsilon_i) = \times$$

In such case, we want $\epsilon_\ominus$ to be *the biggest* and $\epsilon_\oplus$ to be *the smallest* possible, so we would like to deduce the substitution S such that $S\epsilon_\ominus = S\epsilon_1 = \ldots = S\epsilon_n = S\epsilon_\oplus$, as it guarantees that it is in fact the case.

```
solve_within $\Gamma$ $\tau/\varepsilon$ $C$ $S$ =
  $V :=$ gather_free_vars $S\Gamma$ $S\tau/S\varepsilon$
  while $\exists(I_1 * \epsilon_1 <: I_2 * \epsilon_2) \in S\,C.\, \epsilon_1 \neq \epsilon_2 \wedge V(\epsilon_1), V(\epsilon_2)$ matches
      | $\times,\oplus$ | $\ominus,\times$ | $\times,\odot$ | $\odot,\times$
      | $\oplus,\oplus$ | $\ominus,\ominus$ | $\odot,\odot$ $\to$
          $C := C \setminus \{I_1 * \epsilon_1 <: I_2 * \epsilon_2\}$;
          $S :=$(expand $(I_1 \setminus I_2)$ $\epsilon_2$ $S$)$[\epsilon_1 \mapsto I_2 * \epsilon_2]$;
      | $\ominus,\oplus$ | $\ominus,\odot$ | $\odot,\oplus$ $\to$
          $C := C \setminus \{I_1 * \epsilon_1 <: I_2 * \epsilon_2\}$;
          $S :=$(expand $(I_1 \setminus I_2)$ $\epsilon_2$ $S$)$[\epsilon_1 \mapsto I_2 * \epsilon_2]$;
          $V := V[\epsilon_2 \mapsto \odot]$
  for $(\epsilon, \oplus) \in V$ where $\epsilon \notin S\Gamma$:
      $S := S[\epsilon \mapsto \iota]$
  return $C, S$
```

We conclude the description of algorithm by defining function *solve* from $\vdash_{\mathbf{Gen}}$:

$$\begin{aligned} \text{solve}(\Gamma, \tau/\varepsilon, C, S) &= \texttt{solve\_within}\ \Gamma\ \tau/\varepsilon\ C'\ S' \\ &\quad \textit{where}\ S', C' = \texttt{solve\_simple}\ C\ S \end{aligned}$$

4.4 Illustrative example

No what we have a complete picture of how the inference algorithm works, let's take a look how constraints for this complicated term would be generated and resolved.

$$\begin{array}{l}
\textbf{let}\ \textit{compose} = \lambda f.\, \lambda g.\, \lambda x.\ f\ (g\ x)\ \textbf{in} \\
\textbf{let}\ \textit{update} = \lambda(s : \textbf{State}\ t).\, \textbf{Put}_s(f\ (\textbf{Get}_s\ ()))\ \textbf{in} \\
\textbf{handle}_a \\
\quad \textbf{handle}_e \\
\qquad (\lambda x.\, \textbf{Raise}_e\ x)((\textit{update}\ a)(\textit{compose}\ (\lambda x.\, 21)\ (\lambda x.\, x))) \\
\quad \{[(\textbf{Raise}\ (),\ k.\ 37)]; \textbf{return}\ x.\ x\} \\
\{[(\textbf{Get}(),\ k.\ \lambda c.\ kcc); (\textbf{Put}\ v\ k.\ \lambda c.\ (k())v)]; \textbf{return}\ .\ \lambda c.\ x\}
\end{array}$$

We will begin by inferring type for *compose*: *Rozpisywanie tego w ten sposób wygląda mało sensownie, lepiej pewnie byłoby rozpisać to w bardziej programistycznej* *formie, z wcięciami*

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} \dfrac{}{\Gamma \vdash_{\mathbf{Gen}} f \ : t_f/\iota \leadsto \emptyset; id} \\ \dfrac{\dfrac{}{\Gamma \vdash_{\mathbf{Gen}} g \ : t_g/\iota \leadsto \emptyset; id} \qquad \dfrac{}{\Gamma \vdash_{\mathbf{Gen}} x \ : t_x/\iota \leadsto \emptyset; id}}{\Gamma \vdash_{\mathbf{Gen}} (g\ x) \ : t_1/\epsilon_1 \leadsto \{\iota <: \epsilon_1, \iota <: \epsilon_1, t_g <: t_x \to_{\epsilon_1} t_1\}; id} \\ C_1 = \{\iota <: \epsilon_1, \iota <: \epsilon_1, t_g <: t_x \to_{\epsilon_1} t_1, \iota <: \epsilon_2, \epsilon_1 <: \epsilon_2, t_f <: t_1 \to_{\epsilon_2} t_2\} \end{array}}{\Gamma = (f : t_f), (g : t_g), (x : t_x) \quad \Gamma \vdash_{\mathbf{Gen}} f\ (g\ x) \ : t_2/\epsilon_2 \leadsto C_1; id}}{(f : t_f), (g : t_g) \vdash_{\mathbf{Gen}} \lambda x.\ f\ (g\ x) \ : t_x \to_{\epsilon_2} t_2/\iota \leadsto C_1; id}}{(f : t_f) \vdash_{\mathbf{Gen}} \lambda g.\ \lambda x.\ f\ (g\ x) \ : t_g \to t_x \to_{\epsilon_2} t_2/\iota \leadsto C_1; id}}{\vdash_{\mathbf{Gen}} \lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x) \ : t_f \to t_g \to t_x \to_{\epsilon_2} t_2/\iota \leadsto C_1; id}$$

```
solve_simple $C_1$ $id =$
 solve_simple $\{\epsilon_1 <: \epsilon_2, t_g <: t_x \to_{\epsilon_1} t_1, t_f <: t_1 \to_{\epsilon_2} t_2\}$ $id =$
   $\{\epsilon_1 <: \epsilon_2\}; S_1$ where $S_1 = [t_g \mapsto t_x \to_{\epsilon_1} t_1, t_f \mapsto t_1 \to_{\epsilon_2} t_2]$
```

```
solve_within $\emptyset$ $((t_1 \to_{\epsilon_2} t_2) \to (t_x \to_{\epsilon_1} t_1) \to t_x \to_{\epsilon_2} t_2)$ $\iota$ $\{\epsilon_1 <: \epsilon_2\}$ $S_1 =$
  $V = [\epsilon_1 \mapsto \ominus, \epsilon_2 \mapsto \odot];$
return $\emptyset; S_1[\epsilon_1 \mapsto \epsilon_2]$
```

$\text{solve}(\emptyset, t_f \to t_g \to t_x \to_{\epsilon_2} t_2/\iota, C_1, id) =$
$\quad \emptyset; S_1[\epsilon_1 \mapsto \epsilon_2]$

$\text{gen}(\emptyset, (t_1 \to_{\epsilon_2} t_2) \to (t_x \to_{\epsilon_2} t_1) \to t_x \to_{\epsilon_2} t_2) =$
$\quad \forall \alpha, \beta, \gamma, \delta.\ (\alpha \to_\delta \beta) \to (\gamma \to_\delta \alpha) \to \gamma \to_\delta \beta$

Chapter 5

Implementation

Pure OCaml.

5.1 Representation

How calculus, type system, constraints and substitution are implemented.

5.2 Project structure

Which code does what

5.3 Tutorial

Some examples and how to run it.

Chapter 6

Future work

Let-polymorphism allows us to omit implicit type-lambdas (usually denoted by Λ) and type instantiation, making programmers' lives easier. Way of doing so for instance lambdas is yet to be found.

The way we resolve effect constraints is not a formally sound method. Presence of subtyping relation in type inference does not impose much added difficulty from the theoretical point of view, but it does make the practical constraint solving a much harder task. Because of this relation, constraints describing effects form a partially ordered set, and finding the most general substitution that satisfies it, is beyond this work. Proving this property is essential to arguing about the principal type property.

A few works had been written on this topic, proving its difficulty, including the PhD thesis of François Pottier[6]. Although our subtyping relation is much simpler than the one in his work, then maybe, with just the right amount of depth, studying the topological properties of subtyping graph, could lead to some simpler, yet *sound* solution.

The *heuristic* approach we described, produced desired results for all the examples we tried. However, it may be possible that there is type and constraint set that our method fails to generate the *most general unifier*, but given short time window for this work we were not able to find such example.

Bibliography

[1] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (January 2020), 29 pages. `https://doi.org/10.1145/3371116`

[2] Pottier François and Didier Rémy. 2005. The Essence of ML Type Inference. MIT press, Chapter 10 of *Advanced topics in types and programming languages.*

[3] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (*POPL '82*). Association for Computing Machinery, New York, NY, USA, 207–212. `https://doi.org/10.1145/582153.582176`

[4] Robin Milner, A theory of type polymorphism in programming, Journal of Computer and System Sciences, Volume 17, Issue 3, 1978, Pages 348-375, ISSN 0022-0000, `https://doi.org/10.1016/0022-0000(78)90014-4`.

[5] J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. Transactions of the American Mathematical Society, 146, 29-60. `https://doi.org/10.2307/1995158`

[6] François Pottier. Type Inference in the Presence of Subtyping: from Theory to Practice. [Research Report] RR-3483, INRIA. 1998. ffinria-00073205f