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

6 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

Type inference is a process of generating types for expression. It is a processing of answering to the question "what type is this term"? It is present in many mainstream programming languages in some form or other. Most popular type inference algorithm, called $\mathbb{W}$ was presented by Milner[4] and blah blah.

In every day programming side effects occur often, be it writing to memory, performing I/O or mutating state.

Type systems like Hindley-Milner or System-F describe types of pure terms concisely, but programming languages have side effects that they heavily use and those systems cannot describe them. A lot of code is impure blah blah, and while there's lambda calculus which is great mathematical model for pure computation, it cannot do effects. 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, like OCaml, some have their unique way of expressing side effects, like Haskell and its monadic actions Finally there are so called type/effect systems, which have recently gained scholars' attention.

One could think of effects as a generalization of exceptions: calling an operator corresponds to throwing an exception, and enclosing expression by 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 *come back inside* the handled expression.

We transform calculus from to Hindley-Milner world. We present 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].

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

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 h, in which x gets bounds to value of e_x. Supplying the *continuation* k with some value v continues evaluation of e, with v substituted in place oh operation call. But there's nothing special about k, in a sense it's just a function, and programmer may use it in many different ways or not even use it all, returning a value straight from the handler code. After e is evaluated, its value is bound to x in e_r expression, which is the final value returned by handler.

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

The semantics of calculus strictly follows rules defined in Biernacki et al[1]. For formal reduction rules etc see their work.

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.

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

Regarding polymorphism, we only allow prenex polymorphism*universal variables* α, which are quantified by $\forall$ in so called *type schemes* (denoted π). 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.

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

The *type constructor* $\to$ is used to type abstractions, where type $\tau_1 \to_\varepsilon \tau_2$ is given to functions that when applied with input of type τ_1, produce some output of type τ_2, possibly causing effect ε.

For functions, any effects occurring in their body is “hanged” under arrow type, meaning applying an argument to the function would cause some effects to occur.

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

Accordingly, effect of application combines effects of: computing left hand term, effect that “hangs” on it’s arrow type, and the effect of computing the right hand term.

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

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

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. Here 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 different instances of same effect, i.e two cells of memory of **State** τ, there are lambda terms, which can be applied with instances bound by handlers or other instance lambdas. .

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

Then the operators of instance a and type $\tau_1 \to \tau_2$ if applied with some expression e of type τ_1 and effect ε are typed with $\tau_2/a \circ \varepsilon$

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

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

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

Typing handlers:

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

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

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.

Typing operators:

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

3.1 Subtyping

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

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.

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

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

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: $\mathbf{Int} \to \mathbf{Int}$, but clearly any correct type we would think of would not be more general than $\forall \alpha . \alpha \to \alpha$.

In our type system, π is a principal type of e in environments Γ, Θ if

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

For details about subtyping and principal type, see future work. For now we will give an example of how to approximate principal type. Consider function composition, expressed in our calculus as term $\mathit{compose} = \lambda f.\, \lambda g.\, \lambda x.\, f(g x)$ in empty environment, and two type schemes that could be assigned to it:

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

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, $\mathit{compose}\, f\, g$ 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:

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

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 expression's *AST*, generating sub-expressions' types, effects and constraints,

2. Constraint solving: the algorithm builds a substitution that satisfies all the constraints, while generating the most general type.

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

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 rules are somewhat *algorithmic*, meaning the focus shifts from *checking* to *obtaining* a type. If we think about effects as sets of instances, then the subtyping relation of effects simply boils down to set inclusion.

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 other does 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,.

4.2 Generating constraints

As in *algorithm* $\mathbb{W}$, in order to infer type for 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 we shall interpret it.

Judgement $\vdash_{Gen}$ is constructed in a very *algorithmic* way, meaning the focus shifts from *checking* typing derivation to *gathering* constraints and type.

$$\textbf{constraints} \ni C ::= \emptyset \mid \{\tau <: \tau\} \mid \{\varepsilon <: \varepsilon\} \mid C \cup C$$

A substitution is a mapping from type and effect *unification variables* to inferred types or effects, respectively.

$$\textbf{substitution} \ni S ::= id \mid [t \mapsto \tau] \mid [\epsilon \mapsto \varepsilon] \mid S \circ S \mid$$

We will write $S[t \mapsto \tau]$ for $[t \mapsto \tau] \circ S$. Substitution id is simply an identity function.

We treat this $\vdash_{\mathbf{Gen}}$ judgement as a set of rules for *generating* types and constraints (working bottom-up) rather than a type checker ($\vdash$ working top-down). Ideally, for any term e we would like to have that generation would implies syntactic type soundness

$$\Gamma; \Theta \vdash_{\mathbf{Gen}} e \ : \tau/\varepsilon \rightsquigarrow C; S \implies \text{solve}(\Gamma, C, S) = \emptyset; S' \implies S'\Gamma; S'\Theta \vdash e \ : S'\tau/S'\varepsilon$$

We do not prove it in this work, but leave it for future work.

We abstract solving constraints C in environment Γ (under substitution S) to a high-level function *solve*, which returns a reduced set of constraints C' and a new substitution S'.

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

$$\frac{(x:\pi)\in\Gamma \quad \mathrm{inst}(\pi)=\tau}{\Gamma;\Theta \vdash_{\mathbf{Gen}} x \ : \tau/\iota \rightsquigarrow \emptyset; S_{id}} \qquad \frac{\Gamma,(x:t);\Theta \vdash_{\mathbf{Gen}} e \ : \tau/\varepsilon \rightsquigarrow C; S \quad \mathrm{fresh}(t)}{\Gamma;\Theta \vdash_{\mathbf{Gen}} \lambda x.\, e \ : t \to_\varepsilon \tau/\iota \rightsquigarrow C; S}$$

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

$$\frac{\Gamma;\Theta \vdash_{\mathbf{Gen}} e_1 \ : \tau_1/\varepsilon_1 \rightsquigarrow C_1; S_1 \quad S_1\Gamma;S_1\Theta \vdash_{\mathbf{Gen}} e_2 \ : \tau_2/\varepsilon_2 \rightsquigarrow C_2; S_2 \quad \mathrm{fresh}(t) \quad \mathrm{fresh}(\epsilon)}{\Gamma;\Theta \vdash_{\mathbf{Gen}} e_1\, e_2 \ : t/\epsilon \rightsquigarrow C_1\cup C_2\cup\{\varepsilon_1 <: \epsilon, \varepsilon_2 <: \epsilon, \tau_1 <: \tau_2 \to_\epsilon t\}; S_2 S_1}$$

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

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

$$\frac{\begin{array}{c}\Gamma;\Theta,(a:\sigma) \vdash_{\mathbf{Gen}} e \ : \tau/\varepsilon \rightsquigarrow C';S' \\ \mathrm{solve'}(\Gamma,C',S')=C;S \quad S\varepsilon=\iota \quad S\tau'=\tau\end{array}}{\Gamma;\Theta \vdash_{\mathbf{Gen}} \lambda(a:\sigma).e \ : \forall(a:\sigma).\tau/\iota \rightsquigarrow C;S}$$

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

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

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

And now typing handlers:

$$\frac{\Gamma,(x:\mathbf{Unit}),(k:t\to_\varepsilon\tau);\Theta \vdash_{\mathbf{Gen}} e \ : \tau_{\mathbf{Raise}}/\varepsilon_{\mathbf{Raise}} \rightsquigarrow C;S \quad \mathrm{fresh}(t)}{\Gamma;\Theta \vdash_{\mathbf{Gen}} [(\mathbf{Raise},x,k.e)] \ : \mathbf{Error}\triangleright\tau/\varepsilon \rightsquigarrow C\cup\{\tau_{\mathbf{Raise}} <: \tau, \varepsilon_{\mathbf{Raise}} <: \varepsilon\};S}$$

$$\frac{\begin{array}{c}\Gamma,(x:\mathbf{Unit}),(k:\tau'\to_\varepsilon\tau);\Theta \vdash_{\mathbf{Gen}} e_{\mathbf{Get}} \ : \tau_{\mathbf{Get}}/\varepsilon_{\mathbf{Get}} \rightsquigarrow C_{\mathbf{Get}};S_{\mathbf{Get}} \\ S_{\mathbf{Get}}\Gamma,(x:\tau'),(k:\mathbf{Unit}\to_\varepsilon\tau);S_{\mathbf{Get}}\Theta \vdash_{\mathbf{Gen}} e_{\mathbf{Put}} \ : \tau_{\mathbf{Put}}/\varepsilon_{\mathbf{Put}} \rightsquigarrow C_{\mathbf{Put}};S_{\mathbf{Put}} \\ C=C_{\mathbf{Get}}\cup\{\tau_{\mathbf{Get}} <: \tau, \varepsilon_{\mathbf{Get}} <: \varepsilon\}\cup C_{\mathbf{Put}}\cup\{\tau_{\mathbf{Put}} <: \tau, \varepsilon_{\mathbf{Put}} <: \varepsilon\}\end{array}}{\Gamma;\Theta \vdash_{\mathbf{Gen}} [(\mathbf{Get},x,k.e_{\mathbf{Get}});(\mathbf{Put},x,k.e_{\mathbf{Put}})] \ : \mathbf{State}\ \tau'\triangleright\tau/\varepsilon \rightsquigarrow C;S_{\mathbf{Put}}\circ S_{\mathbf{Get}}}$$

It is important that we "return" not only constraints but a substituion also becouse some constraints may have been already resolved in the subtree of term and we need to take it into account. For example, consider term

$$\lambda x.\ \mathbf{let}\ y = x\ 1\ \mathbf{in}\ y$$

// draw subtree

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 like ordinary HM algorithm in its' simplest form. 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) occuring in Γ, τ and ε. The interesting constrains are of form $\epsilon_1 <: I * \epsilon_2$, as there are many substitions 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 *irreducible* by `solve_simple_contraints` are of form $\epsilon_1 <: I * \epsilon_2$; other constraints are either solved or reduced to the *interesting* form.

```
let expand ϵ I S=
  if I = ∅ then S
  else S[ϵ ↦ I ∗ ϵ′] where fresh(ϵ′)

let solve_simple C S =
  match C with
  | ∅            → ∅; S
  | {τ1 <: τ2} ∪ C  →
    match S[τ1], S[τ2] with
    | τ1′, τ2′ when τ1′ = τ2′ →
      solve_simple C S
    | t, τ
    | τ, t →
      solve_simple C S[t ↦ τ]
    | τ1′ →ε1 τ1″, τ2′ →ε2 τ2″ →
```

```
    solve_simple {τ'_1 <: τ'_2, ε_1 <: ε_2, τ''_1 <: τ''_2} ∪ C  S
| {ε_1 <: ε_2} ∪ C  →
  match S[ε_1], S[ε_2] with
  | ι, _ → solve_simple C S
  | I_1, I_2
  | I_1, I_2 ∗ ϵ →
    solve_simple C (expand (I_1 \ I_2) ϵ S)
  | I_1 ∗ ϵ_1, I_2 ∗ ϵ_2 →
    if ϵ_1 = ϵ_2 then solve_simple C (expand (I_1 \ I_2) ϵ_2 S)
    else let C';S' = solve_simple C (expand (I_1 \ I_2) ϵ_2 S)
          in {ϵ_1 <: I_2 ∗ ϵ_2} ∪ C';S'
  | I_1 ∗ ϵ <: I_2 when I_1 ⊆ I_2→
    let C';S' = solve_simple C S
      in {ϵ <: I_2} ∪ 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 $\{\epsilon_1 := I_1\}$ or $\{\epsilon_1 := I_1 * \epsilon_2\}$ obviously resolves the constraint, but clearly some substitutions are better than others.

As discussed in previous chapter, $\rightarrow$ 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` $\Gamma\ \tau/\varepsilon$ we keep information about *variance* of effect unification variables as a function V.

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

If variable appears in Γ, τ, and ε only in *covariant* (*positive*) positions, then $V(\epsilon) = \oplus$. If it appears only *covariantly* (*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

$$\tau[\varepsilon'_\ominus/\varepsilon_\ominus] <: \tau$$

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 explicetly, but often do form *chains* of subtyping constraints like

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

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

$$\epsilon_{\ominus} = \epsilon_1 = \ldots = \epsilon_n = \epsilon_{\oplus}$$

as it guarantees that it is in fact the case.

Sketch of algorithm:

$\texttt{solve\_contraints\_within}\ \Gamma\ \ \tau/\varepsilon\ \ C\ \ S\ \texttt{=}$
$\quad V := \texttt{gather\_free\_vars}\ \ S\Gamma\ \ S\tau/S\varepsilon$
$\quad \texttt{while}\ \exists(I_1 * \epsilon_1 <: I_2 * \epsilon_2) \in S\,C.\ \epsilon_1 \neq \epsilon_2 \wedge V(\epsilon_1), V(\epsilon_2)\ \texttt{matches}$
$\qquad\quad \texttt{|}\ \times,\oplus\ \texttt{|}\ \ominus,\times\ \texttt{|}\ \times,\odot\ \texttt{|}\ \odot,\times$
$\qquad\quad \texttt{|}\ \oplus,\oplus\ \texttt{|}\ \ominus,\ominus\ \texttt{|}\ \odot,\odot\ \to$
$\qquad\qquad C := C \setminus \{I_1 * \epsilon_1 <: I_2 * \epsilon_2\}\,\texttt{;}$
$\qquad\qquad S := S[\epsilon_1 \mapsto I_2 * \epsilon_2]\,\texttt{;}$
$\qquad\quad \texttt{|}\ \ominus,\oplus\ \texttt{|}\ \ominus,\odot\ \texttt{|}\ \odot,\oplus\ \to$
$\qquad\qquad C := C \setminus \{I_1 * \epsilon_1 <: I_2 * \epsilon_2\}\,\texttt{;}$
$\qquad\qquad S := S[\epsilon_1 \mapsto I_2 * \epsilon_2]\,\texttt{;}$
$\qquad\qquad V := V[\epsilon_2 \mapsto \odot]$
$\quad \texttt{for}\ (\epsilon, \oplus) \in V\texttt{:}$
$\qquad S := S[\epsilon \mapsto \iota]$
$\quad \texttt{return}\ C, S$

Finally, we can define function solve' from $\vdash_{\mathbf{Gen}}$:

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

4.4 Illustrative example

As now we know how *solve* functions work, let's take a look how constraints for term would be generated and resolved.

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-polymorhpism 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.

Ensuring that the infered type is principal would most probably require a long and difficult proof. The way we resolve effect constraints is also a heuristic and not a formal method. Such constraints form a graph which topology should be studied with according depth. There are works by Francois that do this in different subtyping relation.

Finally, resolving constraints *formally* would require us to study the topology of graphs constructed. Such constraints form a partially ordered set and finding *formally sound* substitution that satisfies it is beyond this work, so we present a *heuristic* approach, which produces 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* type, 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