A Type Inference System Based on Saturation of Subtyping Constraints*

[Research paper, extended abstract]

Benoît Vaugon and Michel Mauny

LIX, École Polytechnique – U2IS, ENSTA ParisTech
Université Paris-Saclay
828 bd des Maréchaux, 91762 Palaiseau cedex France

Abstract. We present a technique for defining type inference algorithms that involve subtyping, and whose soundness can be proved. We consider an ML-like base language, that can be extended with high-level features such as polymorphism, overloading, variants and pattern-matching, or generalized algebraic data types (GADTs).

This paper presents a powerful and flexible technique for defining type inference algorithms that involve subtyping, and whose soundness can be proved. We consider an ML-like base language, that can be extended with high-level features such as polymorphism, overloading, variants and pattern-matching, or generalized algebraic data types (GADTs).

In our formalism, a typing algorithm is defined as a set of inference rules of three distinct forms: *typing* rules collect subtyping constraints to be satisfied, *instanciation* rules instanciate type schemes, and *saturation* rules specify how to check the validity and consistency of collected constraints. Essentially, type inference then proceeds in two intertwinned phases: one that extracts constraints and the other one that saturates the sets of constraints.

Types are always manipulated in a special “flat” form: the grammar of types is not recursive, and distinguishes types for building values, that occur at the left of a subtyping constraint from those that occur at the right, for deconstructing values. The simplicity of our type algebra simplifies termination proofs.

Constraints are represented as logical formulae involving conjunction, disjunction and negation, that connect subtype relations between two types. We never “solve” constraints: instead, we simply check their consistency by saturation. This way, we avoid the classical indecidability problems, and allow for more expressive constraints and more powerful type inference.

We choose to keep the type language as simple as possible. When adding a new feature in our language, we extend the constraint language, rather than the type language, preserving the good properties of the latter. An advantage of this formalism is that our saturation mechanism does not need variance information on type constructors: the initial orientation of constraints that comes from the typing rules associated to a given type constructor is sufficient to encode variance. Furthermore, mutable data are easily dealt with, even in presence of polymorphism.

The general formalism that we define here enables the encoding of rather powerful language constructs such as overloading, a refinement of the typing of pattern-matching, an extension of ML-like polymorphism, as well as a complete type inference algorithm for GADTs with subtyping [BV16]. None of these extensions are described in this extended abstract. We expect the full paper to give hints on the treatment of those features. For a full treatment of all those features (overloading excepted), we refer the interested reader to the first author’s PhD manuscript [BV16].

* This work is part of the first author’s PhD thesis [BV16], defended on March 15, 2016.

1 Introduction

The presentation of type synthesis of a program as the collection of constraints to be satisfied by its sub-expressions, followed by their resolution, is now classical. To cite only a few of them, [MM82, SP05, PJ06] consider and solve equality constraints, and [TS96, G11, DM15] consider subtyping constraints.

In this work, we follow a "collect-and-saturate" approach in the spirit of [TS96], rather than "collect-and-solve". The main differences between our approach and [TS96] is that we have a much simpler type language and a richer constraint language. Moreover, we use a uniform formalism that allows us to both represent typing proofs and effectively implement a type inference algorithm. Our formalism is not really original: it is well known that inference rules can be used to define functions (syntax-directed rules, with a clear distinction between parameters and return values). Still, using it to define complete inference algorithms and to prove their soundness is original, as far as we know.

This extended abstract is organized as follows: section 2 presents the programming language that we consider, section 3 present the type algebra and the constraint language, and section 4 introduce our base type inference system. Section 5 states the properties of this system and section 6 briefly describes its implementation. An annex lists the full set of inference rules.

2 The language

The language that we consider here is given in Figure 1: it is a functional language in the spirit of ML, with constants and primitive operations. The language has also data constructors ($\mathtt{K}_i$) and pattern-matching, with an optional default case.

Syntax	
$\mathtt{e ::=}$	
$\mid \mathtt{x} \mid \lambda\ \mathtt{x}\ .\ \mathtt{e} \mid \mathtt{e}_1\ \mathtt{e}_2$	*λ-calculus*
$\mid \mathtt{c}$	*constants*
$\mid (\mathtt{e}_1,\ \mathtt{e}_2)$	*pairs*
$\mid \mathtt{p}^1\ \mathtt{e} \mid \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2$	*primitive operations, including projections*
$\mid \mathtt{K}\ \mathtt{e}$	*data constructors*
$\mid \mathtt{let}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$	*local declarations*
$\mid \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$	*conditional*
$\mid \mathtt{match}\ \mathtt{e}\ \mathtt{with}\ \mathtt{K}_1\ \mathtt{x}_1 \rightarrow \mathtt{e}_1 \parallel \ldots \parallel \mathtt{K}_n\ \mathtt{x}_n \rightarrow \mathtt{e}_n$	*pattern-matching*
$\mid \mathtt{match}\ \mathtt{e}\ \mathtt{with}\ \mathtt{K}_1\ \mathtt{x}_1 \rightarrow \mathtt{e}_1 \parallel \ldots \parallel \mathtt{K}_n\ \mathtt{x}_n \rightarrow \mathtt{e}_n \parallel \mathtt{x}_d \rightarrow \mathtt{e}_d$	*... with a default case*

Fig. 1. The expression language

The language has a classical call-by-value semantics[1].

3 Types and constraints

Types are distinguished according to whether they come from building values (we call them "left-types", that will occur at the left of subtyping constraints) or from value deconstruction ("right-types") such as pattern-matching or function application. Right-types include types of the form $\{\ \mathrm{K}_1\ \alpha_1 \parallel \ldots\ \}$ which correspond to deconstruction of variants by pattern-matching.

[1] Note that the evaluation order does not interfere with typing, and lazyness would not complicate the treatment that is given here.

$$\begin{array}{rl} \tau^l & ::= \alpha \mid (\alpha_1,\ \ldots,\ \alpha_n)\ \texttt{t} \mid \texttt{K}\ \alpha \\ \tau^r & ::= \alpha \mid (\alpha_1,\ \ldots,\ \alpha_n)\ \texttt{t} \\ & \mid \{\ \texttt{K}_1\ \alpha_1 \parallel \ldots \parallel \texttt{K}_n\ \alpha_n\ \} \\ & \mid \{\ \texttt{K}_1\ \alpha_1 \parallel \ldots \parallel \texttt{K}_n\ \alpha_n \parallel \alpha_d\ \} \end{array}$$

Types

$$\begin{array}{rl} \texttt{C} & ::= \tau^l \leqslant \tau^r \mid \tau^l \nleqslant \tau^r \\ \Psi & ::= \texttt{C}_1 \vee \ldots \vee \texttt{C}_n \\ \Phi & ::= \Psi_1 \wedge \ldots \wedge \Psi_n \end{array}$$

Constraints

$$\sigma ::= [\ \forall\alpha_1 \ldots \alpha_n\ .\ \alpha \mid \Phi\ ]$$

Type schemes

$$\Gamma ::= (\texttt{x}_1,\ \sigma_1),\ \ldots,\ (\texttt{x}_n,\ \sigma_n)$$

Typing environments

Fig. 2. Types and constraints

Typing environments are equipped with the two classical operations: one for adding a new binding $(\texttt{x},\ \sigma)$ in a type environment Γ, written $\Gamma \oplus (\texttt{x},\ \sigma)$, and the other one to extract the type scheme associated to a variable $\texttt{x}$ in Γ, written $\Gamma[\texttt{x}]$.

Constraints can be direct ($\leqslant$) or negated ($\nleqslant$), the latter occurring for instance when dealing with precise typing of pattern matching (not presented in this paper, see [BV16]). A conjunction Φ, which we sometimes call a constraint set, is made of disjunctions Ψ. When we write a disjunction as $\Psi \vee C$, we call Ψ the *alternative* to C.

Types and constraints are given in figure 2.

4 Inference systems

4.1 Inference rules

In our formalism, an inference system has three different kinds of rules, namely *typing* rules, that we sometimes call "T-rules", *instanciation* rules (I-rules), and *saturation* rules (S-rules).

Typing rules. The typing rules (whose names start with a "T") collect constraints. There is usually exactly one such rule per syntax construct. T-rules have the following shape:

$$\text{Txxx}\ \frac{\ldots \qquad \ldots \qquad \ldots}{\Phi, \Gamma \vdash \Psi \vee \texttt{e} : \alpha \rhd \Phi'}$$

Their conclusion should be read as "*under the set of constraints Φ, in the environment Γ:*

- *either e can be of type α producing constraints which when saturated with Φ generate Φ',*
- *or the disjunction Ψ is valid and compatible with Φ, and the saturation of $\Phi \wedge \Psi$ generates Φ'*".

The set Φ' is the enrichment of Φ that is produced when the type synthesis of e has been performed. Of course, when $e : \alpha$, Φ' constraints α, and the "type" of e can be expressed as "α such that Φ'".

Saturation rules. Their names start with a "S" have the following shapes:

$$\text{Sxxx}\ \frac{\ldots \qquad \ldots \qquad \ldots}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi'}$$

They should be read as: "*either $\tau^l \leqslant \tau^r$ is valid and compatible with Φ, or the disjunction Ψ is non-empty and compatible with Φ; Φ' being the resulting set of constraints*".

Instanciation rules. Similar to S-rules, I-rules have the shape:

$$\text{Ixxx}\ \frac{\dots \quad \dots \quad \dots}{\Phi \vdash \Psi \vee \sigma \leqslant \tau^r \rhd \Phi'}$$

and compare a type scheme σ with a type, instead of a comparing two types as S-rules do.

In the following, we call "T-node" (resp. I-node, S-node) a node that is an instance of a T-rule (resp. I-rule, S-rule).

Structure of inference trees. A successful type inference produces an inference tree composed of three successive layers. The lower layer (in green on figure 3) is built from typing rules, and is therefore isomorphic to the program. The middle layer is made of instanciation rules (in red): they are used when some primitive have type schemes instead of simple types, or when the language has constructs whose typing generates polymorphism. Finally, the topmost layer is made of saturation rules (in blue).

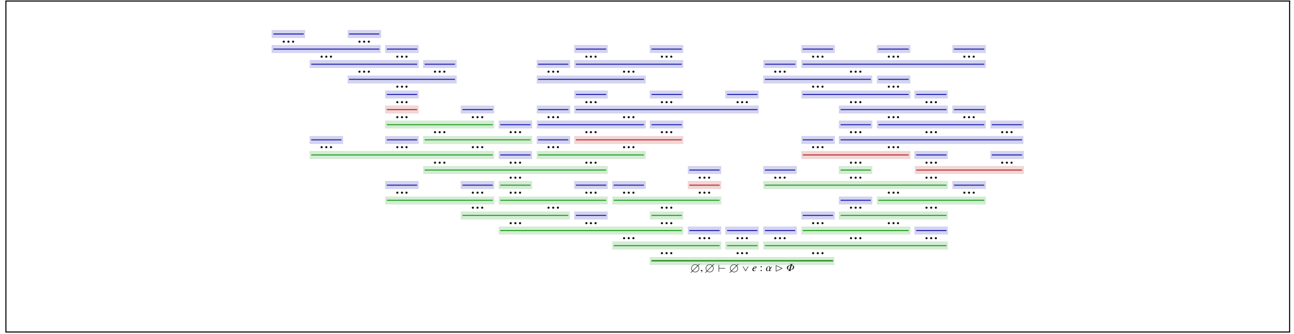

Fig. 3. The different layers of an inference tree

4.2 Type schemes

Before giving examples of inference rules, and expressing the properties of our system, we need a few definitions and notations about type schemes.

Definition 1 (Order relation on type schemes). *Two type schemes* $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ *and* $[\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *satisfy* $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ] \leqslant [\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *if there exists a substitution R of variables to variables (like a renaming, but not necessarily injective) such that:*

- $R(\alpha_0) = \alpha'_0$
- $R(\{\ \alpha_1, \dots, \alpha_n\ \}) \subset \{\ \alpha'_1, \dots, \alpha'_{n'}\ \}$
- $\forall\alpha\ .\ \alpha \notin \{\ \alpha_1, \dots, \alpha_n\ \} \Rightarrow R(\alpha) = \alpha$
- $R(\Phi) \subset \Phi'$

Intuitively, when we have $\sigma_1 \leqslant \sigma_2$, a value of type σ_1 can be used where a value of type σ_2 is expected. In other words, σ_1 is a subtype of σ_2.

Generalization We need a generalization function, which, when given a typing environment, generalizes a type variable together with its constraints. Here is its definition:

$$\textsc{gen}(\alpha, \Phi, \Gamma) \;\triangleq\; [\, \forall(\textsc{ftv}(\Phi) \backslash \textsc{ftv}(\Gamma)) \,.\, \alpha \mid \Phi \,]$$

Typability of an expression Given a particular inference system, we define a relation (_ : _) that relates an expression e and a type scheme σ:

Definition 2 **($e : \sigma$).** *We write $e : \sigma$ if, when given a type variable α, the two following properties hold:*

- *there exists Φ such that we have a proof tree of $\emptyset, \emptyset \vdash \emptyset \vee e : \alpha \rhd \Phi$*
- $\textsc{gen}(\alpha, \Phi, \emptyset) \leqslant \sigma$

We say that e is *typable* if there exists σ such that $e : \sigma$. Note that if $e : \sigma$, then $e : \sigma'$ for any σ' supertype of σ.

4.3 Examples of inference rules

The T-rule for a conditional `if` e_1 `then` e_2 `else` e_3 generates a right-type `bool` (in deconstruction position) for the test e_1, and propagates the constraints from the two branches e_2 and e_3 to the result type α:

TIf

$$\frac{\begin{array}{c}\texttt{let}\ \alpha'\ \textit{fresh} \\ \Phi_1 \vdash \Psi \vee \alpha' \leqslant \texttt{bool} \rhd \Phi_2 \\ \Phi_2, \Gamma \vdash \Psi \vee e_1 : \alpha' \rhd \Phi_3 \qquad \Phi_3, \Gamma \vdash \Psi \vee e_2 : \alpha \rhd \Phi_4 \qquad \Phi_4, \Gamma \vdash \Psi \vee e_3 : \alpha \rhd \Phi_5\end{array}}{\Phi_1, \Gamma \vdash \Psi \vee \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \alpha \rhd \Phi_5}$$

Abstractions build functional values, this is the reason why the T-rule for abstractions constraints the resulting type α to the left with an arrow type:

TLambda

$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi_1, \Gamma \oplus x : \alpha_1 \vdash \Psi \vee e : \alpha_2 \rhd \Phi_2 \qquad \Phi_2 \vdash \Psi \vee \alpha_1 \to \alpha_2 \leqslant \alpha \rhd \Phi_3}{\Phi_1, \Gamma \vdash \Psi \vee \lambda\, x \,.\, e : \alpha \rhd \Phi_3}$$

S-rules decompose constraints and saturate them by transitivity of subtyping. For this, we define auxiliary functions that extract components of constraint sets. As an example, the $\textsc{rights}$ function extracts from Φ all right-types and alternatives Ψ associated to a given α:

$$\textsc{rights}(\alpha,\ \Phi) \;\triangleq\; \{\, (\Psi, \tau^r) \mid (\Psi \vee \alpha \leqslant \tau^r) \in \Phi \,\}$$

When adding a new disjunction $\Psi \vee \tau^l \leqslant \alpha$, we extract from Φ all right-types τ^r bigger than α, and their alternatives Ψ', and we generate disjunctions $\tau^l \leqslant \tau^r$, with $\Psi \vee \Psi'$ as alternative:

STransRight

$$\frac{\begin{array}{c}\texttt{let}\ (\Psi_1, \tau^r_1), \ldots, (\Psi_n, \tau^r_n) = \textsc{rights}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \Psi \vee \Psi_1 \vee \tau^l \leqslant \tau^r_1 \rhd \Phi_2 \qquad \cdots \qquad \Phi_n \vdash \Psi \vee \Psi_n \vee \tau^l \leqslant \tau^r_n \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \Psi \vee \tau^l \leqslant \alpha \rhd \Phi_{n+1}}$$

When a τ^l of the form $\mathrm{K}\ \alpha$, corresponding to the type inference of the application of a data constructor K, has to be compared to a $\{\, \ldots,\ \mathrm{K}\ \alpha',\ \ldots \,\}$, coming from typing a pattern-matching accepting values built with K, we propagate the subtype relation to type variables associated to the arguments of the data constructor:

SVariantMatch

$$\frac{\Phi_1 \vdash \Psi \vee \alpha \leqslant \alpha' \rhd \Phi_2}{\Phi_1 \vdash \Psi \vee \mathrm{K}\,\alpha \leqslant \{\, \ldots,\ \mathrm{K}\,\alpha',\ \ldots \,\} \rhd \Phi_2}$$

The saturation of constraints checks the validity of constraints (see the definitions below). When the validity of a constraint set cannot be checked, the type inference fails and a type error is reported.

Definition 3 (Validity of a subtyping constraint C). *A subtyping constraint C is said to be* valid *if there exists a saturation rule whose conclusion has the form $\Phi \vdash \Psi \vee C \rhd \Phi'$, that is, if it is possible to perform at least one saturation step from $\Phi \vdash \Psi \vee C \rhd \Phi'$.*

In other words, a subtyping constraint is valid if it has one of the following forms:

- $\alpha \leqslant (\alpha'_1, \ldots, \alpha'_n)\ t$
- $(\alpha_1, \ldots, \alpha_n)\ t \leqslant \alpha'$
- $\alpha \leqslant \{\ \ldots, K\ \alpha', \ldots\ \}$
- $K\ \alpha \leqslant \alpha'$
- $\alpha \leqslant \alpha'$
- $(\alpha_1, \ldots, \alpha_n)\ t \leqslant (\alpha'_1, \ldots, \alpha'_n)\ t$
- $K\ \alpha \leqslant \{\ \ldots, K\ \alpha', \ldots\ \}$

Definition 4 (Validity of a disjunction of subtyping constraints Ψ). *A disjunction of subtyping constraints Ψ is valid if at least one of its members (a subtyping constaint) is valid.*

Definition 5 (Validity of a conjunction of subtyping constraints Φ). *A conjunction of subtyping relations Φ is valid if all its members are valid.*

Definition 6 (Saturation of a constraint set Φ). *A set of constraints Φ is said to be* saturated *if it satisfies all the following properties:*

► *Comparison of parameterized types with the same name:*

- $\forall\ \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_n\ .$
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t}) \in \Phi \Rightarrow$
  $(\Psi \vee \alpha_1 \leqslant \alpha'_1) \in \Phi \wedge \ldots \wedge (\Psi \vee \alpha_n \leqslant \alpha'_n) \in \Phi \wedge$
  $(\Psi \vee \alpha'_1 \leqslant \alpha_1) \in \Phi \wedge \ldots \wedge (\Psi \vee \alpha'_n \leqslant \alpha_n) \in \Phi$
- $\forall\ \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_n\ .$
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \nleqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t}) \in \Phi \Rightarrow$
  $(\Psi \vee \alpha_1 \nleqslant \alpha'_1 \vee \ldots \vee \alpha_n \nleqslant \alpha'_n \vee \alpha'_1 \nleqslant \alpha_1 \vee \ldots \vee \alpha'_n \nleqslant \alpha_n) \in \Phi$

► *When a disjunction member is invalid, the rest must be present in Φ:*

- $\forall\ \Psi, \alpha\ .$
  $(\Psi \vee \alpha \nleqslant \alpha) \in \Phi \Rightarrow \Psi \in \Phi$
- $\forall\ \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_p, \mathtt{u}\ .$
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_p)\ \mathtt{u}) \in \Phi \Rightarrow \Psi \in \Phi$

► *Transitivity through an α:*

- $\forall\ \Psi_1, \tau^l, \alpha, \Psi_2, \tau^r\ .$
  $(\Psi_1 \vee \tau^l \leqslant \alpha) \in \Phi \wedge (\Psi_2 \vee \alpha \leqslant \tau^r) \in \Phi \Rightarrow (\Psi_1 \vee \Psi_2 \vee \tau^l \leqslant \tau^r) \in \Phi$
- $\forall\ \Psi_1, \alpha, \tau^l, \Psi_2, \tau^r\ .$
  $(\Psi_1 \vee \alpha \leqslant \tau^r) \in \Phi \wedge (\Psi_2 \vee \alpha \nleqslant \tau^l) \in \Phi \Rightarrow (\Psi_1 \vee \Psi_2 \vee \tau^r \nleqslant \tau^l) \in \Phi$
- $\forall\ \Psi_1, \tau^l, \alpha, \Psi_2, \tau^r\ .$
  $(\Psi_1 \vee \tau^l \leqslant \alpha) \in \Phi \wedge (\Psi_2 \vee \tau^r \nleqslant \alpha) \in \Phi \Rightarrow (\Psi_1 \vee \Psi_2 \vee \tau^r \nleqslant \tau^l) \in \Phi$

The saturation of a set of constraints computes the smallest saturated set that contains it.

4.4 Example

In order to examplify the usage of these rules, let us give for instance the "typing part" of the inference tree of the expression (`not true`):

$$
\dfrac{
\dfrac{
\begin{array}{cc}
\dfrac{\nabla}{\varnothing \vdash \varnothing \vee \alpha_5 \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_1} & \dfrac{\nabla}{\Phi_1 \vdash \varnothing \vee \alpha_6 \leqslant \mathtt{bool} \rhd \Phi_2} \\
\dfrac{\nabla}{\Phi_2 \vdash \varnothing \vee \mathtt{bool} \leqslant \alpha_7 \rhd \Phi_3} & \dfrac{\nabla_{\mathtt{T}}}{\Phi_3 \vdash \varnothing \vee \alpha_6 \to \alpha_7 \leqslant \alpha_5 \rhd \Phi_{15}}
\end{array}
}{\varnothing \vdash \varnothing \vee \sigma \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_{15}}\,\textsc{Inst}
\quad
\dfrac{\nabla}{\Phi_{15} \vdash \varnothing \vee \alpha_4 \leqslant \alpha \rhd \Phi_{16}}
\quad
\dfrac{
\dfrac{
\dfrac{\nabla}{\Phi_{16} \vdash \varnothing \vee \alpha_8 \leqslant \alpha_3 \rhd \Phi_{17}} \quad \dfrac{\nabla}{\Phi_{17} \vdash \varnothing \vee \mathtt{bool} \leqslant \alpha_8 \rhd \Phi_{18}}
}{\Phi_{16} \vdash \varnothing \vee [\, \forall \alpha_0 \,.\, \alpha_0 \mid \mathtt{bool} \leqslant \alpha_0 \,] \leqslant \alpha_3 \rhd \Phi_{18}}\,\textsc{Inst}
}{\Phi_{16}, \varnothing \vdash \varnothing \vee \mathtt{true} : \alpha_3 \rhd \Phi_{18}}\,\textsc{TConst}
}{\varnothing, \varnothing \vdash \varnothing \vee \mathtt{not\ true} : \alpha \rhd \Phi_{18}}\,\textsc{TApplyPrim1}
$$

with:

- $\sigma = [\, \forall \alpha_0\, \alpha_1\, \alpha_2 \,.\, \alpha_0 \mid \alpha_1 \leqslant \mathtt{bool} \wedge \mathtt{bool} \leqslant \alpha_2 \wedge \alpha_1 \to \alpha_2 \leqslant \alpha_0 \,]$

- $\Phi_1 = \alpha_5 \leqslant \alpha_3 \to \alpha_4$
- $\Phi_2 = \Phi_1 \wedge \alpha_6 \leqslant \mathtt{bool}$
- $\Phi_3 = \Phi_2 \wedge \mathtt{bool} \leqslant \alpha_7$
- $\Phi_4 = \Phi_3 \wedge \alpha_6 \to \alpha_7 \leqslant \alpha_5$
- $\Phi_5 = \Phi_4 \wedge \alpha_6 \to \alpha_7 \leqslant \alpha_3 \to \alpha_4$
- $\Phi_6 = \Phi_5 \wedge \alpha_6 \leqslant \alpha_3$
- $\Phi_7 = \Phi_6 \wedge \alpha_3 \leqslant \alpha_6$
- $\Phi_8 = \Phi_7 \wedge \alpha_3 \leqslant \mathtt{bool}$
- $\Phi_9 = \Phi_8 \wedge \alpha_3 \leqslant \alpha_3$
- $\Phi_{10} = \Phi_9 \wedge \alpha_6 \leqslant \alpha_6$
- $\Phi_{11} = \Phi_{10} \wedge \alpha_7 \leqslant \alpha_4$
- $\Phi_{12} = \Phi_{11} \wedge \mathtt{bool} \leqslant \alpha_4$
- $\Phi_{13} = \Phi_{12} \wedge \alpha_4 \leqslant \alpha_7$
- $\Phi_{14} = \Phi_{13} \wedge \alpha_4 \leqslant \alpha_4$
- $\Phi_{15} = \Phi_{14} \wedge \alpha_7 \leqslant \alpha_7$
- $\Phi_{16} = \Phi_{15} \wedge \alpha_4 \leqslant \alpha \wedge \alpha_7 \leqslant \alpha \wedge \mathtt{bool} \leqslant \alpha$
- $\Phi_{17} = \Phi_{15} \wedge \alpha_8 \leqslant \alpha_3 \wedge \alpha_8 \leqslant \alpha_6 \wedge \alpha_8 \leqslant \mathtt{bool}$
- $\Phi_{18} = \Phi_{17} \wedge \mathtt{bool} \leqslant \alpha_8 \wedge \mathtt{bool} \leqslant \alpha_3 \wedge \mathtt{bool} \leqslant \alpha_6$

where the subtree $\nabla\!\texttt{T}$ is:

$$
\text{SNewConstraint}\ \dfrac{\text{SDiffVar}\ \dfrac{}{\Phi_6 \vdash \varnothing \vee \alpha_6 \leq \alpha_3 \rhd \Phi_6}}{\Phi_5 \vdash \varnothing \vee \alpha_6 \leqslant \alpha_3 \rhd \Phi_6}
\qquad
\text{SNewConstraint}\ \dfrac{\text{SDiffVar}\ \dfrac{\text{SNewConstraint}\ \dfrac{\text{STransLeft}\ \dfrac{}{\Phi_8 \vdash \varnothing \vee \alpha_3 \leq \texttt{bool} \rhd \Phi_8}}{\Phi_7 \vdash \varnothing \vee \alpha_3 \leqslant \texttt{bool} \rhd \Phi_8} \quad \text{SNewConstraint}\ \dfrac{\text{SSameVar}\ \dfrac{}{\Phi_9 \vdash \varnothing \vee \alpha_3 \leq \alpha_3 \rhd \Phi_9}}{\Phi_8 \vdash \varnothing \vee \alpha_3 \leqslant \alpha_3 \rhd \Phi_9} \quad \text{SNewConstraint}\ \dfrac{\text{SSameVar}\ \dfrac{}{\Phi_{10} \vdash \varnothing \vee \alpha_6 \leq \alpha_6 \rhd \Phi_{10}}}{\Phi_9 \vdash \varnothing \vee \alpha_6 \leqslant \alpha_6 \rhd \Phi_{10}}}{\Phi_7 \vdash \varnothing \vee \alpha_3 \leq \alpha_6 \rhd \Phi_{10}}}{\Phi_6 \vdash \varnothing \vee \alpha_3 \leqslant \alpha_6 \rhd \Phi_{10}}
$$

$$
\text{SNewConstraint}\ \dfrac{\text{SDiffVar}\ \dfrac{\text{SNewConstraint}\ \dfrac{\text{STransRight}\ \dfrac{}{\Phi_{12} \vdash \varnothing \vee \texttt{bool} \leq \alpha_4 \rhd \Phi_{12}}}{\Phi_{11} \vdash \varnothing \vee \texttt{bool} \leqslant \alpha_4 \rhd \Phi_{12}}}{\Phi_{11} \vdash \varnothing \vee \alpha_7 \leq \alpha_4 \rhd \Phi_{12}}}{\Phi_{10} \vdash \varnothing \vee \alpha_7 \leqslant \alpha_4 \rhd \Phi_{12}}
\qquad
\text{SNewConstraint}\ \dfrac{\text{SDiffVar}\ \dfrac{\text{SNewConstraint}\ \dfrac{\text{SSameVar}\ \dfrac{}{\Phi_{14} \vdash \varnothing \vee \alpha_4 \leq \alpha_4 \rhd \Phi_{14}}}{\Phi_{13} \vdash \varnothing \vee \alpha_4 \leqslant \alpha_4 \rhd \Phi_{14}} \quad \text{SNewConstraint}\ \dfrac{\text{SSameVar}\ \dfrac{}{\Phi_{15} \vdash \varnothing \vee \alpha_7 \leq \alpha_7 \rhd \Phi_{15}}}{\Phi_{14} \vdash \varnothing \vee \alpha_7 \leqslant \alpha_7 \rhd \Phi_{15}}}{\Phi_{13} \vdash \varnothing \vee \alpha_4 \leq \alpha_7 \rhd \Phi_{15}}}{\Phi_{12} \vdash \varnothing \vee \alpha_4 \leqslant \alpha_7 \rhd \Phi_{15}}
$$

$$
\text{SNewConstraint}\ \dfrac{\text{STransRight}\ \dfrac{\text{SNewConstraint}\ \dfrac{\text{STypeConstr}\ \dfrac{}{\Phi_5 \vdash \varnothing \vee \alpha_6 \to \alpha_7 \leq \alpha_3 \to \alpha_4 \rhd \Phi_{15}}}{\Phi_4 \vdash \varnothing \vee \alpha_6 \to \alpha_7 \leqslant \alpha_3 \to \alpha_4 \rhd \Phi_{15}}}{\Phi_4 \vdash \varnothing \vee \alpha_6 \to \alpha_7 \leq \alpha_5 \rhd \Phi_{15}}}{\Phi_3 \vdash \varnothing \vee \alpha_6 \to \alpha_7 \leqslant \alpha_5 \rhd \Phi_{15}}
$$

5 Properties

Theorem 1 (Termination). *For a given finite expression e, the type inference of e, that is, building the inference tree for* $\varnothing, \varnothing \vdash \varnothing \vee e : \alpha \rhd \Phi$*, following the given inference rules, always terminates.*

Since each inference rule builds exactly one node of the inference tree, proving that inference trees are always finite suffices to prove termination.

First, let us note that the placement of T-nodes and S-nodes is not arbitrary: the root of the tree is always a T-node, and no S-rule has a T-rule as premise. Consequently, there exists a bottom part of the tree that contains only T-nodes, and which forms itself a tree whose leaves are either I-nodes or S-nodes.

In the following, we name "saturation subtree" (or S-subtree) a maximal subtree composed only of S-nodes.

The only premises of I-nodes are conclusions of S-rules: the premises of I-nodes are thus also roots of S-subtrees.

Sketch of the proof. In order to prove that all inference trees have a finite size, we start by showing that the "typing part" of such a tree is finite, then that it contains only a finite number of I-nodes, and then that all S-subtrees have a finite size.

Theorem 2 (Soundness). *For any expression e and type scheme* σ *such that* $e : \sigma$*, one of the following properties holds:*

- *evaluating e does not terminate,*
- *e evaluates to a value v and* $v : \sigma$*.*

The detailed proof of this theorem can be found in [BV16], and will be sketched in the full version of this paper.

6 Implementation

The formalism that we have used for defining our inference systems can receive a direct implementation. Performing the type inference for an expression *e* only needs to build the inference tree by using the rules of the inference system under consideration.

At each step of building the tree, either no rule can be applied, and the type inference stops, raising an error, or there is exactly one rule that applies, and type inference goes on, deterministically. This property is true for the "typing part" (building T-nodes), as well as for the "saturation part" (building S-nodes). Furthermore, each rule precisely mentions the type variables to be generated, and there is no (implicit) global operation to perform, such as, for instance, renaming of type variables.

It is therefore extremely easy to extract a working implementation of an inference algorithm from such a type inference system. Of course, proceeding this way results in an implementation much less efficient than unification-based systems that use mutable data structures to represent type variables, and the famous union-find algorithm (*cf.* [CP15, H91]). Our systems simply cannot use unification, and we cannot use those tools directly to deal with our subtyping constraints.

The fact is that such a direct implementation of our systems produces an extremely inefficient resulting type inference program. The reason is threefold:

1. disjunctions are problematic: they are the source of combinatorial explosions in the saturation mechanisms, because of backtracking;
2. the generalization mechanism naively encapsulates the whole set of constraints and provokes an explosion of the number of type variables generated at each instanciation, as well as an explosion of the number of constraints;

3. the saturation mechanism keeps in Φ constraints that are consequences of others and generates an explosion of the number of items returned by LEFTS and RIGHTS, resulting in useless constraints and computations.

Another difficulty with this naïve implementation technique lies in the readibility of the type schemes provided to the user: the size of scheme becomes quickly huge (for the reasons above), and a simple clean up by analyzing dependencies is practically insufficient for having readable results.

However, we found two effective ways to improve the performance of our type inference prototype:

- clean up the sets of constraints contained in type schemes. For doing this, we have developed three orthogonal techniques:
  - improve dependencies analysis when generalizing types: this improvement allows for an aggressive management of dependencies between several disjunctions of constraints;
  - reinforcement of saturation, considering all disjunctions "modulo rotation";
  - aggressive clean up of constraint sets, using a detection of "equivalent constraints subsets";
- use a dedicated representation of constraint sets containing disjunctions: this representation allows for optimizing the primitives that are used by cleaning algorithms, and limits certain redundant computations during saturation.

These techniques have been tested and seem to work well in practice. They bring gains of several orders of magnitude in computing time as well as in the size of constraint sets. The cleaning mechanisms provide us with a further advantage in improving the readibility of type schemes.

7 Conclusion

We have presented in this extended abstract a inference system based of saturation of subtyping constraints, providing terminating and sound inference algorithms. This system has been successfully extended to perform type inference of the following language features:

- overloading (with dynamic dispatch)
- more precise typing of pattern matching (that keeps the relationship between input patterns and output results)
- a generalization of ML polymorphism that enables polymorphic usages of function arguments
- type inference for GADTs with subtyping

The full paper will give hints on how these extensions work. All but overloading are fully described in the first author's PhD thesis [BV16].

8 Annex

8.1 A complete inference system

Meta-functions

$$\textsc{lefts}(\alpha,\ \Phi) \triangleq \{\ (\Psi, \tau^l) \mid (\Psi \vee \tau^l \leqslant \alpha) \in \Phi\ \}$$
$$\textsc{rights}(\alpha,\ \Phi) \triangleq \{\ (\Psi, \tau^r) \mid (\Psi \vee \alpha \leqslant \tau^r) \in \Phi\ \}$$
$$\overline{\textsc{lefts}}(\alpha,\ \Phi) \triangleq \{\ (\Psi, \tau^r) \mid (\Psi \vee \tau^r \nleqslant \alpha) \in \Phi\ \}$$
$$\overline{\textsc{rights}}(\alpha,\ \Phi) \triangleq \{\ (\Psi, \tau^l) \mid (\Psi \vee \alpha \nleqslant \tau^l) \in \Phi\ \}$$

La meta-fonction T associe un schéma de type aux constantes et aux primitives. Par exemple:

- $T(3) \triangleq [\ \forall\alpha\ .\ \alpha \mid \texttt{int} \leqslant \alpha\ ]$
- $T(\texttt{not}) \triangleq [\ \forall\alpha\alpha_1\alpha_2\ .\ \alpha \mid \alpha_1 \to \alpha_2 \leqslant \alpha\ \wedge\ \alpha_1 \leqslant \texttt{bool}\ \wedge\ \texttt{bool} \leqslant \alpha_2\ ]$

Typing

TConst
$$\frac{\Phi \vdash \Psi \vee T(\texttt{c}) \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \Psi \vee \texttt{c} : \alpha \rhd \Phi'}$$

TApplyPrim1
$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi \vdash \Psi \vee T(\texttt{p}^1) \leqslant \alpha_1 \to \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_2 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha_1 \rhd \Phi'''}{\Phi, \Gamma \vdash \Psi \vee \texttt{p}^1\ \texttt{e}_1 : \alpha \rhd \Phi'''}$$

TApplyPrim2
$$\frac{\begin{array}{c}\texttt{let}\ \alpha_0, \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha_0 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_3 \leqslant \alpha \rhd \Phi'' \\ \Phi'' \vdash \Psi \vee T(\texttt{p}^2) \leqslant \alpha_1 \to \alpha_0 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha_1 \rhd \Phi'''' \qquad \Phi'''', \Gamma \vdash \Psi \vee \texttt{e}_2 : \alpha_2 \rhd \Phi'''''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{p}^2\ \texttt{e}_1\ \texttt{e}_2 : \alpha \rhd \Phi'''''}$$

TVar
$$\frac{\texttt{when}\ \Gamma[\texttt{x}]\ \textit{defined} \qquad \Phi \vdash \Psi \vee \Gamma[\texttt{x}] \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \Psi \vee \texttt{x} : \alpha \rhd \Phi'}$$

TLambda
$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \oplus (\texttt{x}, \alpha_1) \vdash \Psi \vee \texttt{e} : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_1 \to \alpha_2 \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \lambda\,\texttt{x}\,.\,\texttt{e} : \alpha \rhd \Phi''}$$

TApp
$$\frac{\begin{array}{c}\texttt{let}\ \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \\ \Phi' \vdash \Psi \vee \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha_1 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \texttt{e}_2 : \alpha_2 \rhd \Phi''''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{e}_1\ \texttt{e}_2 : \alpha \rhd \Phi''''}$$

TPair

$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha_1 \rhd \Phi' \qquad \Phi', \Gamma \vdash \Psi \vee \texttt{e}_2 : \alpha_2 \rhd \Phi'' \qquad \Phi'' \vdash \Psi \vee \alpha_1 \times \alpha_2 \leqslant \alpha \rhd \Phi'''}{\Phi, \Gamma \vdash \Psi \vee (\texttt{e}_1,\ \texttt{e}_2) : \alpha \rhd \Phi'''}$$

TConstr

$$\frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \texttt{e} : \alpha' \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \texttt{K}\ \alpha' \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \texttt{K e} : \alpha \rhd \Phi''}$$

TIf

$$\frac{\begin{array}{c}\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha' \leqslant \texttt{bool} \rhd \Phi' \\ \Phi', \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha' \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \texttt{e}_2 : \alpha \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \texttt{e}_3 : \alpha \rhd \Phi''''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 : \alpha \rhd \Phi''''}$$

TLet

$$\frac{\texttt{let}\ \alpha'\ \textit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \texttt{e}_1 : \alpha' \rhd \Phi' \qquad \Phi', \Gamma \oplus (\texttt{x}, \textsc{gen}(\alpha', \Phi', \Gamma)) \vdash \Psi \vee \texttt{e}_2 : \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \texttt{let x} = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 : \alpha \rhd \Phi''}$$

TMatch

$$\frac{\begin{array}{c}\texttt{let}\ \alpha_e, \alpha_1, \ldots, \alpha_n\ \textit{fresh} \\ \Phi \vdash \Psi \vee \alpha_e \leqslant \{\ \texttt{K}_1\ \alpha_1\ \parallel\ \ldots\ \parallel\ \texttt{K}_n\ \alpha_n\ \} \rhd \Phi_0 \qquad \Phi_0, \Gamma \vdash \Psi \vee \texttt{e} : \alpha_e \rhd \Phi_1 \\ \Phi_1, \Gamma \oplus (\texttt{x}_1,\ \alpha_1) \vdash \Psi \vee \texttt{e}_1 : \alpha \rhd \Phi_2 \\ \ldots \\ \Phi_n, \Gamma \oplus (\texttt{x}_n,\ \alpha_n) \vdash \Psi \vee \texttt{e}_n : \alpha \rhd \Phi_{n+1}\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{match e with}\ \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1\ \parallel\ \ldots\ \parallel\ \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n : \alpha \rhd \Phi_{n+1}}$$

TMatchDefault

$$\frac{\begin{array}{c}\texttt{let}\ \alpha_e, \alpha_1, \ldots, \alpha_n, \alpha_d\ \textit{fresh} \\ \Phi \vdash \Psi \vee \alpha_e \leqslant \{\ \texttt{K}_1\ \alpha_1\ \parallel\ \ldots\ \parallel\ \texttt{K}_n\ \alpha_n\ \parallel\ \alpha_d\ \} \rhd \Phi_0 \qquad \Phi_0, \Gamma \vdash \Psi \vee \texttt{e} : \alpha_e \rhd \Phi_1 \\ \Phi_1, \Gamma \oplus (\texttt{x}_1,\ \alpha_1) \vdash \Psi \vee \texttt{e}_1 : \alpha \rhd \Phi_2 \\ \ldots \\ \Phi_n, \Gamma \oplus (\texttt{x}_n,\ \alpha_n) \vdash \Psi \vee \texttt{e}_n : \alpha \rhd \Phi_{n+1} \\ \Phi_{n+1}, \Gamma \oplus (\texttt{x}_d,\ \alpha_d) \vdash \Psi \vee \alpha_e \leqslant \{\ \texttt{K}_1\ \alpha_1\ \parallel\ \ldots\ \parallel\ \texttt{K}_n\ \alpha_n\ \} \vee \texttt{e}_d : \alpha \rhd \Phi_{n+2}\end{array}}{\Phi, \Gamma \vdash \Psi \vee \texttt{match e with}\ \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1\ \parallel\ \ldots\ \parallel\ \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n\ \parallel\ \texttt{x}_d \to \texttt{e}_d : \alpha \rhd \Phi_{n+2}}$$

Instanciation

Inst

$$\frac{\begin{array}{c}\texttt{let}\ \alpha'_1, \ldots, \alpha'_n\ \textit{fresh} \quad \Phi \vdash \Psi \vee \alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^{n} \leqslant \tau^r \rhd \Phi_1 \\ \Phi_1 \vdash \Psi \vee \Psi_1[\alpha_i \mapsto \alpha'_i]_{i=1}^{n} \rhd \Phi_2 \quad \cdots \quad \Phi_p \vdash \Psi \vee \Psi_p[\alpha_i \mapsto \alpha'_i]_{i=1}^{n} \rhd \Phi_{p+1}\end{array}}{\Phi \vdash \Psi \vee [\ \forall \alpha_1\ \ldots\ \alpha_n\ .\ \alpha_0 \mid \Psi_1 \wedge \cdots \wedge \Psi_p\ ] \leqslant \tau^r \rhd \Phi_{p+1}}$$

Saturation

$$\frac{\texttt{when}\ (\varPsi \vee \tau^l \leqslant \tau^r) \notin \varPhi \qquad \varPhi \wedge (\varPsi \vee \tau^l \leqslant \tau^r) \vdash^{?} \varPsi \vee \tau^l \leqslant \tau^r \rhd \varPhi'}{\varPhi \vdash \varPsi \vee \tau^l \leqslant \tau^r \rhd \varPhi'}\ \textsc{SNewConstraint}(\leqslant)$$

$$\frac{\texttt{when}\ (\varPsi \vee \tau^l \leqslant \tau^r) \in \varPhi}{\varPhi \vdash \varPsi \vee \tau^l \leqslant \tau^r \rhd \varPhi}\ \textsc{SAlreadyProved}(\leqslant)$$

$$\frac{\texttt{when}\ (\varPsi \vee \tau^l \nleqslant \tau^r) \notin \varPhi \qquad \varPhi \wedge (\varPsi \vee \tau^l \nleqslant \tau^r) \vdash^{?} \varPsi \vee \tau^l \nleqslant \tau^r \rhd \varPhi'}{\varPhi \vdash \varPsi \vee \tau^l \nleqslant \tau^r \rhd \varPhi'}\ \textsc{SNewConstraint}(\nleqslant)$$

$$\frac{\texttt{when}\ (\varPsi \vee \tau^l \nleqslant \tau^r) \in \varPhi}{\varPhi \vdash \varPsi \vee \tau^l \nleqslant \tau^r \rhd \varPhi}\ \textsc{SAlreadyProved}(\nleqslant)$$

$$\frac{}{\varPhi \vdash \varPsi \vee \alpha \leqslant \alpha \rhd \varPhi}\ \textsc{SLeqSameVar}$$

$$\frac{\texttt{when}\ \texttt{t} \neq \texttt{u}}{\varPhi \vdash \varPsi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \nleq (\alpha'_1, \ldots, \alpha'_p)\ \texttt{u} \rhd \varPhi}\ \textsc{SNotLeqDiffParamed}$$

$$\frac{\varPhi \vdash \{\ \varPsi \vee \alpha_i \leqslant \alpha'_i\ \}_{i=1}^n \rhd \varPhi' \qquad \varPhi' \vdash \{\ \varPsi \vee \alpha'_i \leqslant \alpha_i\ \}_{i=1}^n \rhd \varPhi''}{\varPhi \vdash \varPsi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \leq (\alpha'_1, \ldots, \alpha'_n)\ \texttt{t} \rhd \varPhi''}\ \textsc{SLeqSameParamed}$$

$$\frac{\varPhi \vdash \varPsi\ \vee\ \alpha_1 \nleq \alpha'_1 \vee \cdots \vee \alpha_n \nleq \alpha'_n\ \vee\ \alpha'_1 \nleq \alpha_1 \vee \cdots \vee \alpha'_n \nleq \alpha_n \rhd \varPhi'}{\varPhi \vdash \varPsi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \nleq (\alpha'_1, \ldots, \alpha'_n)\ \texttt{t} \rhd \varPhi'}\ \textsc{SNotLeqSameParamed}$$

$$\frac{\varPhi \vdash \varPsi \vee \phi \rhd \varPhi'}{\varPhi \vdash \varPsi \vee \phi \vee \alpha \nleq \alpha \rhd \varPhi'}\ \textsc{SNotLeqSameVar}$$

$$\frac{\varPhi \vdash \varPsi \vee \phi \rhd \varPhi'}{\varPhi \vdash \varPsi \vee \phi \vee (\alpha_1, \ldots, \alpha_n)\ \texttt{t} \leq (\alpha'_1, \ldots, \alpha'_p)\ \texttt{u} \rhd \varPhi'}\ \textsc{SLeqDiffParamed}$$

SVarLeqParamed

$$\frac{\begin{array}{cc}\texttt{let}\ (\Psi_1,\tau^l_1),\ldots,(\Psi_p,\tau^l_p) = \textsc{lefts}(\alpha,\ \Phi) & \texttt{let}\ (\Psi'_1,\tau'^l_1),\ldots,(\Psi'_q,\tau'^l_q) = \overline{\textsc{rights}}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee \tau^l_i \leqslant (\alpha_1,\ldots,\alpha_n)\ \texttt{t}\ \}^p_{i=1} \rhd \Phi' & \Phi' \vdash \{\ \Psi \vee \Psi'_i \vee (\alpha_1,\ldots,\alpha_n)\ \texttt{t} \nleqslant \tau'^l_i\ \}^q_{i=1} \rhd \Phi''\end{array}}{\Phi \vdash \Psi \vee \alpha \leq (\alpha_1,\ldots,\alpha_n)\ \texttt{t} \rhd \Phi''}$$

SParamedLeqVar

$$\frac{\begin{array}{cc}\texttt{let}\ (\Psi_1,\tau^r_1),\ldots,(\Psi_p,\tau^r_p) = \textsc{rights}(\alpha,\ \Phi) & \texttt{let}\ (\Psi'_1,\tau'^r_1),\ldots,(\Psi'_q,\tau'^r_q) = \overline{\textsc{lefts}}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee (\alpha_1,\ldots,\alpha_n)\ \texttt{t} \leqslant \tau^r_i\ \}^p_{i=1} \rhd \Phi' & \Phi' \vdash \{\ \Psi \vee \Psi'_i \vee \tau'^r_i \nleqslant (\alpha_1,\ldots,\alpha_n)\ \texttt{t}\ \}^q_{i=1} \rhd \Phi''\end{array}}{\Phi \vdash \Psi \vee (\alpha_1,\ldots,\alpha_n)\ \texttt{t} \leq \alpha \rhd \Phi''}$$

SVarLeqVar

$$\frac{\begin{array}{c}\texttt{when}\ \alpha_1 \neq \alpha_2 \\ \texttt{let}\ (\Psi^l_1,\tau^l_1),\ldots,(\Psi^l_p,\tau^l_p) = \textsc{lefts}(\alpha_1,\ \Phi),\ (\varnothing,\alpha_1) \quad \texttt{let}\ (\Psi'^l_1,\tau'^l_1),\ldots,(\Psi'^l_q,\tau'^l_q) = \overline{\textsc{rights}}(\alpha_1,\ \Phi) \\ \texttt{let}\ (\Psi^r_1,\tau^r_1),\ldots,(\Psi^r_r,\tau^r_r) = \textsc{rights}(\alpha_2,\ \Phi),\ (\varnothing,\alpha_2) \quad \texttt{let}\ (\Psi'^r_1,\tau'^r_1),\ldots,(\Psi'^r_s,\tau'^r_s) = \overline{\textsc{lefts}}(\alpha_2,\ \Phi) \\ \Phi \vdash \{\ \{\ \Psi \vee \Psi^l_i \vee \Psi^r_j \vee \tau^l_i \leqslant \tau^r_j\ \}^p_{i=1}\ \}^r_{j=1} \rhd \Phi' \\ \Phi' \vdash \{\ \{\ \Psi \vee \Psi'^l_i \vee \Psi^r_j \vee \tau^r_j \nleqslant \tau'^l_i\ \}^q_{i=1}\ \}^r_{j=1} \rhd \Phi'' \quad \Phi'' \vdash \{\ \{\ \Psi \vee \Psi^l_i \vee \Psi'^r_j \vee \tau'^r_j \nleqslant \tau^l_i\ \}^p_{i=1}\ \}^s_{j=1} \rhd \Phi'''\end{array}}{\Phi \vdash \Psi \vee \alpha_1 \leq \alpha_2 \rhd \Phi'''}$$

SVarNotLeqParamed

$$\frac{\texttt{let}\ (\Psi_1,\tau^r_1),\ldots,(\Psi_p,\tau^r_p) = \textsc{rights}(\alpha,\ \Phi) \qquad \Phi \vdash \{\ \Psi \vee \Psi_i \vee \tau^r_i \nleqslant (\alpha_1,\ldots,\alpha_n)\ \texttt{t}\ \}^p_{i=1} \rhd \Phi'}{\Phi \vdash \Psi \vee \alpha \nleq (\alpha_1,\ldots,\alpha_n)\ \texttt{t} \rhd \Phi'}$$

SParamedNotLeqVar

$$\frac{\texttt{let}\ (\Psi_1,\tau^l_1),\ldots,(\Psi_p,\tau^l_p) = \textsc{lefts}(\alpha,\ \Phi) \qquad \Phi \vdash \{\ \Psi \vee \Psi_i \vee (\alpha_1,\ldots,\alpha_n)\ \texttt{t} \nleqslant \tau^l_i\ \}^p_{i=1} \rhd \Phi'}{\Phi \vdash \Psi \vee (\alpha_1,\ldots,\alpha_n)\ \texttt{t} \nleq \alpha \rhd \Phi'}$$

SVarNotLeqVar

$$\frac{\begin{array}{c}\texttt{when}\ \alpha_1 \neq \alpha_2 \\ \texttt{let}\ (\Psi^r_1,\tau^r_1),\ldots,(\Psi^r_p,\tau^r_p) = \textsc{rights}(\alpha_1,\ \Phi),\ (\varnothing,\alpha_1) \quad \texttt{let}\ (\Psi^l_1,\tau^l_1),\ldots,(\Psi^l_q,\tau^l_q) = \textsc{lefts}(\alpha_2,\ \Phi),\ (\varnothing,\alpha_2) \\ \Phi \vdash \{\ \{\ \Psi \vee \Psi^r_i \vee \Psi^l_j \vee \tau^r_i \nleqslant \tau^l_j\ \}^p_{i=1}\ \}^q_{j=1} \rhd \Phi'\end{array}}{\Phi \vdash \Psi \vee \alpha_1 \nleq \alpha_2 \rhd \Phi'}$$

8.2 Outline of termination and soundness proofs

References

[CP15] Arthur Charguéraud and François Pottier. “Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation”. In: ITP ’15 (Interactive Theorem Proving) 9236 (Aug. 2015), pp. 137–153 (cit. on p. 9).

[DM15] Stephen Dolan and Alan Mycroft. “Polymorphism, subtyping and type inference in MLsub”. In: Computer (2015) (cit. on p. 2).

[G11] Eli Gottlieb. “Simple, Decidable Type Inference with Subtyping”. In: CoRR ’11 (Computing Research Repository - arXiv) abs/1104.3116 (2011). URL: http://arxiv.org/abs/1104.3116 (cit. on p. 2).

[H91] Fritz Henglein. “Efficient Type Inference for Higher-Order Binding-Time Analysis”. In: FPCA ’91 (Functional Programming and Computer Architecture) (1991), pp. 448–472 (cit. on p. 9).

[MM82] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: TOPLAS ’82 (Transactions on Programming Languages and Systems) 4.2 (Apr. 1982), pp. 258–282. ISSN: 0164-0925. DOI: 10.1145/357162.357169. URL: http://doi.acm.org/10.1145/357162.357169 (cit. on p. 2).

[PJ06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. “Simple Unification-based Type Inference for GADTs”. In: SIGPLAN Notices 41.9 (Sept. 2006), pp. 50–61. ISSN: 0362-1340. DOI: 10.1145/1160074.1159811. URL: http://doi.acm.org/10.1145/1160074.1159811 (cit. on p. 2).

[SP05] Vincent Simonet and François Pottier. *Constraint-Based Type Inference for Guarded Algebraic Data Types*. Research Report 5462. INRIA, Jan. 2005. URL: http://gallium.inria.fr/~fpottier/publis/simonet-pottier-hmg.ps.gz (cit. on p. 2).

[TS96] Valery Trifonov and Scott Smith. “Subtyping Constrained Types”. In: SAS ’06 (Static Analysis Symposium) 1145 (1996), pp. 349–365 (cit. on p. 2).

[BV16] Benoît Vaugon. “Sous-typage par saturation de contraintes: théorie et implémentation”. PhD thesis. Université Paris-Saclay, 2016 (cit. on pp. 1, 3, 9, 10).