# Extending Hindley-Milner Type Inference with Coercive Subtyping (long version)

Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany

**Abstract.** We investigate how to add coercive structural subtyping to a type system for simply-typed lambda calculus with Hindley-Milner polymorphism. Coercions allow to convert between different types, and their automatic insertion can greatly increase readability of terms. We present a type inference algorithm that, given a term without type information, computes a type assignment and determines at which positions in the term coercions have to be inserted to make it type-correct. The algorithm is sound and, if the subtype relation on base types is a disjoint union of lattices, also complete. Also, a sound but incomplete extension of the algorithm to type classes is given. The algorithm has been implemented in the proof assistant Isabelle.

## 1 Introduction

The main idea of subtype polymorphism, or simply subtyping, is to allow the programmer to omit type conversions, also called *coercions*. Inheritance in object-oriented programming languages can be viewed as a form of subtyping.

Although the ability to omit coercions is important to avoid unnecessary clutter in programs, subtyping is not a common feature in functional programming languages, such as ML or Haskell. The main reason for this is the increase in complexity of type inference systems with subtyping compared to Milner's well-known algorithm W [7]. In contrast, the theorem prover Coq supports coercive subtyping, albeit in an incomplete manner. Our contributions to this extensively studied area are:

– a comparatively simple type and coercion inference algorithm with
– soundness and completeness results improving on related work (see the beginning of §3 and the end of §6), and
– a practical implementation in the Isabelle theorem prover. This extension is very effective, for example, in the area of numeric types (*nat*, *int*, *real* etc), which require coercions that used to clutter up Isabelle text.

Our work does not change the underlying Hindley-Milner type system (and hence leaves the Isabelle kernel unchanged!) but infers where coercions need to be inserted to make some term type correct.

The rest of this paper is structured as follows. In §2 we introduce terms, types, coercions and subtyping. §3 presents our type inference algorithm for simply-typed lambda calculus with coercions and Hindley-Milner polymorphism, while §4 examines the extension of this algorithm to type classes. In §5, we provide a proof of the total correctness of our algorithm. Further, §6 discusses restrictions on the subtype relation that are needed to ensure completeness of the algorithm without type classes and provides a proof of completeness. An outline of related research is given in §7.

## 2 Notation and terminology

### 2.1 Terms and types

The types and terms of simply-typed lambda calculus are given by the following grammars:

$$\tau = \alpha \mid T \mid C\ \tau\ \ldots\ \tau$$

$$t = x \mid c_{[\overline{\alpha} \mapsto \overline{\tau}]} \mid (\lambda x : \tau.\ t) \mid t\ t$$

A type can be a *type variable* (denoted by $\alpha$, $\beta$, ...), a *base type* (denoted by $S$, $T$, $U$, ...), or a *compound type*, which is a type constructor (denoted by $C$, $D$, ...) applied to a list of type arguments. The number of arguments of a type constructor $C$, which must be at least one, is called the *arity* of $C$. The function type is a special case of a binary type constructor. We use the common infix notation $\tau \to \sigma$ in this case. Terms can be variables (denoted by $x$, $y$, ...), abstractions, or applications. In addition, a term can contain *constants* (denoted by $c$, $d$, ...) of polymorphic type. All terms are defined over a *signature* $\Sigma$ that maps each constant to a *schematic type*, i.e. a type containing variables. In every occurrence of a constant $c$, the variables in its schematic type can be instantiated in a different way, for which we use the notation $c_{[\overline{\alpha} \mapsto \overline{\tau}]}$, where $\overline{\alpha}$ denotes the vector of free variables in the type of $c$ (ordered in a canonical way), and $\overline{\tau}$ denotes the vector of types that the free variables are instantiated with. The type checking rules for terms are shown in Figure 1.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Ty-Var} \qquad \frac{\Sigma(c) = \sigma}{\Gamma \vdash c_{[\overline{\alpha} \mapsto \overline{\tau}]} : \sigma[\overline{\alpha} \mapsto \overline{\tau}]}\ \textsc{Ty-Const}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau.\ t : \tau \to \sigma}\ \textsc{Ty-Abs} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ t_2 : \sigma}\ \textsc{Ty-App}$$

**Fig. 1.** Type checking rules

## 2.2 Subtyping and coercions

We write $\tau <: \sigma$ to denote that $\tau$ is a *subtype* of $\sigma$. The subtyping relation that we consider in this paper is *structural*: if $\tau <: \sigma$, then $\tau$ and $\sigma$ can only differ in their base types. For example, we may have $C\,T <: C\,U$, but not $C\,T <: S$. Type checking rules for systems with subtypes are often presented using a so-called *subsumption* rule

$$\frac{\Gamma \vdash t : \tau \qquad \tau <: \sigma}{\Gamma \vdash t : \sigma}$$

allowing a term $t$ of type $\tau$ to be used in a context where a term of the supertype $\sigma$ would be expected. The problem of deciding whether a term is typable using the subsumption rule is equivalent to the problem of deciding whether this term can be made typable without the subsumption rule by inserting coercion functions in appropriate places in the term. Rather than extending our type system with a subsumption rule, we therefore introduce a new judgement $\Gamma \vdash t \rightsquigarrow u : \tau$ that, given a context $\Gamma$ and a term $t$, returns a new term $u$ augmented with coercions, together with a type $\tau$, such that $\Gamma \vdash u : \tau$ holds. We write $\tau <:_c \sigma$ to mean that $c$ is a coercion of type $\tau \to \sigma$. Coercions can be built up from a set of coercions $\mathcal{C}$ between base types, and from a set of *map functions* $\mathcal{M}$ for building coercions between constructed types from coercions between their argument types as shown in Figure 2. The sets $\mathcal{C}$ and $\mathcal{M}$ are parameters of our setup. We restrict $\mathcal{M}$ to contain at most one map function for a type constructor.

**Definition 1 (Map function).** *Let $C$ be an $n$-ary type constructor. A function $f$ of type*

$$\tau_1 \to \cdots \to \tau_n \to C\ \alpha_1\ \ldots\ \alpha_n \to C\ \beta_1\ \ldots\ \beta_n$$

*where $\tau_i \in \{\alpha_i \to \beta_i, \beta_i \to \alpha_i\}$, is called a* map function *for $C$. If $\tau_i = \alpha_i \to \beta_i$, then $C$ is called* covariant *in the $i$-th argument wrt. $f$, otherwise* contravariant.

$$\frac{}{\tau <:_{id} \tau}\ \textsc{Gen-Refl} \qquad \frac{\Sigma(c) = T \to U \qquad c \in \mathcal{C}}{T <:_c U}\ \textsc{Gen-Base}$$

$$\frac{T <:_{c_1} U \qquad U <:_{c_2} S}{T <:_{\lambda x:T.c_2\ (c_1\ x)} S}\ \textsc{Gen-Trans}$$

$$\frac{\begin{array}{c} map_C : (\delta_1 \to \rho_1) \to \cdots \to (\delta_n \to \rho_n) \to C\ \alpha_1 \ldots \alpha_n \to C\ \beta_1 \ldots \beta_n \in \mathcal{M} \\ \theta = \{\overline{\alpha} \mapsto \overline{\tau}, \overline{\beta} \mapsto \overline{\sigma}\} \qquad \forall 1 \le i \le n.\ \theta(\delta_i) <:_{c_i} \theta(\rho_i) \end{array}}{C\ \tau_1 \ldots \tau_n <:_{\theta(map_C\ c_1\ \ldots\ c_n)} C\ \sigma_1 \ldots \sigma_n}\ \textsc{Gen-Cons}$$

**Fig. 2.** Coercion generation

For the implementation of type checking and inference algorithms, the subsumption rule is problematic, because it is not syntax directed. However, it can be shown that any derivation of $\Gamma \vdash t : \sigma$ using the subsumption rule can be transformed into a derivation of $\Gamma \vdash t : \tau$ with $\tau <: \sigma$, in which the subsumption rule is only applied to function arguments [13, §22.8]. Consequently, the coercion insertion judgement shown in Figure 3 only inserts coercions in argument positions of functions by means of the Coerce-App rule.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \tau}\ \textsc{Coerce-Var} \qquad \frac{\Sigma(c) = \sigma}{\Gamma \vdash c_{[\overline{\alpha} \mapsto \overline{\tau}]} \rightsquigarrow c_{[\overline{\alpha} \mapsto \overline{\tau}]} : \sigma[\overline{\alpha} \mapsto \overline{\tau}]}\ \textsc{Coerce-Const}$$

$$\frac{\Gamma, x : \tau \vdash t \rightsquigarrow u : \sigma}{\Gamma \vdash \lambda x : \tau.\ t \rightsquigarrow \lambda x : \tau.\ u : \tau \to \sigma}\ \textsc{Coerce-Abs}$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow u_1 : \tau_{11} \to \tau_{12} \qquad \Gamma \vdash t_2 \rightsquigarrow u_2 : \tau_2 \qquad \tau_2 <:_c \tau_{11}}{\Gamma \vdash t_1\ t_2 \rightsquigarrow u_1\ (c\ u_2) : \tau_{12}}\ \textsc{Coerce-App}$$

**Fig. 3.** Coercion insertion

### 2.3 Type substitutions and unification

A central component of type inference systems is a *unification* algorithm for types. Implementing such an algorithm for the type expressions introduced in §2.1 is straightforward, since this is just an instance of first-order unification. We write *mgu* for the function computing the most general unifier. It produces a *type substitution*, denoted by $\theta$, which is a function mapping type variables to types such that $\theta\alpha \neq \alpha$ for only finitely many $\alpha$. We will sometimes use the notation $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ to denote such substitutions. Type substitutions are extended to types, terms, and any other data structures containing type variables in the usual way. The function *mgu* is overloaded: it can be applied to pairs of terms, where $\theta\tau = \theta\sigma$ if $\theta = mgu(\tau, \sigma)$, to (finite) sets of equality constraints, where $\theta\tau_i = \theta\sigma_i$ if $\theta = mgu\{\tau_1 \doteq \sigma_1, \ldots, \tau_n \doteq \sigma_n\}$, as well as to (finite) sets of types, where $\theta\tau_1 = \cdots = \theta\tau_n$ if $\theta = mgu\{\tau_1, \ldots, \tau_n\}$.

## 3 Type Inference with Coercions

In a system without coercions, *type inference* means to find a type substitution $\theta$ and a type $\tau$ for a given term $t$ and context $\Gamma$ such that $t$ becomes typable, i.e. $\theta\Gamma \vdash \theta t : \tau$. In a system with coercions, type inference also has to insert coercions into the term $t$ in appropriate places, yielding a term $u$ for which $\theta\Gamma \vdash \theta t \rightsquigarrow u : \tau$ and $\theta\Gamma \vdash u : \tau$ holds. A naive way of doing type inference in this setting would be to compute the substitution $\theta$ and insert the coercions on-the-fly, as suggested by Luo [6]. The idea behind Luo's type inference algorithm is to try to do standard Hindley-Milner type inference first, and locally repair typing problems by inserting coercions only if the standard algorithm fails. However, this approach has a serious drawback: the success or failure of the algorithm depends on the order in which the types of subterms are inferred. To see why this is the case, consider the following example.

*Example 1.* Let $\Sigma = \{leq : \alpha \to \alpha \to \mathbb{B}, n : \mathbb{N}, i : \mathbb{Z}\}$ be the signature containing a polymorphic predicate $leq$ (e.g. less-or-equal), as well as a natural number constant $n$ and an integer constant $i$. Moreover, assume that the set of coercions $\mathcal{C} = \{int : \mathbb{N} \to \mathbb{Z}\}$ contains a coercion from natural numbers to integers, but not from integers to natural numbers, since this would cause a loss of information. As shown in Figure 4, the terms $leq_{[\alpha \mapsto \beta]}\ i\ n$ and $leq_{[\alpha \mapsto \beta]}\ n\ i$ can both be made type correct by applying the type substitution $\{\beta \mapsto \mathbb{Z}\}$ and inserting coercions, but the naive algorithm can only infer the type of the first term. Since the term is an application, the algorithm would first infer (using standard Hindley-Milner type inference) that the function denoted by the subterm $leq_{[\alpha \mapsto \beta]}\ i$ has type $\mathbb{Z} \to \mathbb{B}$ with the type substitution $\{\beta \mapsto \mathbb{Z}\}$. Similarly, for the subterm $n$ the type $\mathbb{N}$ is inferred. Since the argument type $\mathbb{Z}$ of the function does not match the type $\mathbb{N}$ of its argument, the algorithm inserts the coercion $int$ to repair the typing problem, yielding the term $leq_{[\alpha \mapsto \mathbb{Z}]}\ i\ (int\ n)$ with type $\mathbb{B}$. In contrast, when inferring the type of the term $leq_{[\alpha \mapsto \beta]}\ n\ i$, the algorithm would first infer that the subterm $leq_{[\alpha \mapsto \beta]}\ n$ has type $\mathbb{N} \to \mathbb{B}$, using the type substitution $\{\beta \mapsto \mathbb{N}\}$. The subterm $i$ is easily seen to have type $\mathbb{Z}$, which does not match the argument type $\mathbb{N}$ of the function. However, in this case, the type mismatch cannot be repaired, since there is no coercion from $\mathbb{Z}$ to $\mathbb{N}$, and so the algorithm fails.

$$\dfrac{\dfrac{\dfrac{\Sigma(leq) = \alpha \to \alpha \to \mathbb{B}}{\Gamma \vdash leq_{[\alpha \mapsto \mathbb{Z}]} \rightsquigarrow leq_{[\alpha \mapsto \mathbb{Z}]} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{B}} \quad \dfrac{\Sigma(i) = \mathbb{Z}}{\Gamma \vdash i \rightsquigarrow i : \mathbb{Z}} \quad \dfrac{}{\mathbb{Z} <:_{id} \mathbb{Z}}}{\Gamma \vdash leq_{[\alpha \mapsto \mathbb{Z}]}\ i \rightsquigarrow leq_{[\alpha \mapsto \mathbb{Z}]}\ i : \mathbb{Z} \to \mathbb{B}} \quad \dfrac{\Sigma(n) = \mathbb{N}}{\Gamma \vdash n \rightsquigarrow n : \mathbb{N}} \quad \dfrac{int : \mathbb{N} \to \mathbb{Z} \in \mathcal{C}}{\mathbb{N} <:_{int} \mathbb{Z}}}{\Gamma \vdash leq_{[\alpha \mapsto \mathbb{Z}]}\ i\ n \rightsquigarrow leq_{[\alpha \mapsto \mathbb{Z}]}\ i\ (int\ n) : \mathbb{B}}$$

$$\dfrac{\dfrac{\dfrac{\Sigma(leq) = \alpha \to \alpha \to \mathbb{B}}{\Gamma \vdash leq_{[\alpha \mapsto \mathbb{Z}]} \rightsquigarrow leq_{[\alpha \mapsto \mathbb{Z}]} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{B}} \quad \dfrac{\Sigma(n) = \mathbb{N}}{\Gamma \vdash n \rightsquigarrow n : \mathbb{N}} \quad \dfrac{int : \mathbb{N} \to \mathbb{Z} \in \mathcal{C}}{\mathbb{N} <:_{int} \mathbb{Z}}}{\Gamma \vdash leq_{[\alpha \mapsto \mathbb{Z}]}\ n \rightsquigarrow leq_{[\alpha \mapsto \mathbb{Z}]}\ (int\ n) : \mathbb{Z} \to \mathbb{B}} \quad \dfrac{\Sigma(i) = \mathbb{Z}}{\Gamma \vdash i \rightsquigarrow i : \mathbb{Z}} \quad \dfrac{}{\mathbb{Z} <:_{id} \mathbb{Z}}}{\Gamma \vdash leq_{[\alpha \mapsto \mathbb{Z}]}\ n\ i \rightsquigarrow leq_{[\alpha \mapsto \mathbb{Z}]}\ (int\ n)\ i : \mathbb{B}}$$

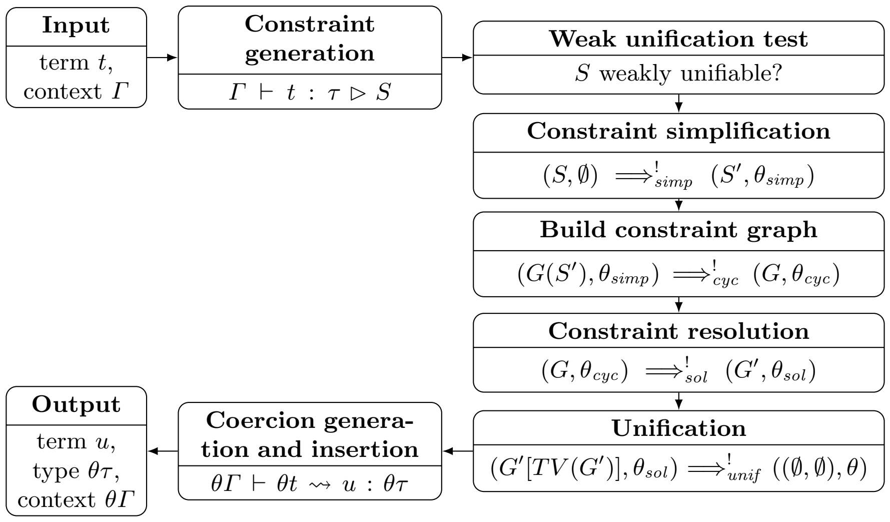
**Fig. 4.** Examples for coercion insertion

The strategy for coercion insertion used in the Coq proof assistant (originally due to Saïbi [15], who provides no soundness or completeness results) suffers from similar problems, which the reference manual describes as the "normal" behaviour of coercions [3, §17.11]. Our goal is to provide a complete algorithm that does not fail in cases such as the above.

### 3.1 Coercive subtyping using subtype constraints

The algorithm presented here generates subtype constraints first, and postpones their solution as well as the insertion of coercions to a later stage of the algorithm. The set of all constraints provides us with a global view on the term that we are processing, and therefore avoids the problems of a local algorithm.

The algorithm can be divided into four major phases. First, we generate subtype constraints by recursively traversing the term. Then, we simplify these constraints, which can be inequalities between arbitrary types, until the constraint set contains only inequalities between base types and variables. The next step is to organize these *atomic* constraints in a graph and solve them, which means to find a type substitution. Applying this substitution to the whole constraint set results in inequalities that are consistent with the given partial order on base types. Finally, the coercions are inserted by traversing the term for the second time. A visualization of the main steps of the algorithm in form of a control flow is shown in Figure 5.



Input
term $t$, context $\Gamma$
Constraint generation
$\Gamma \vdash t : \tau \rhd S$
Weak unification test
$S$ weakly unifiable?
Constraint simplification
$(S, \emptyset) \Longrightarrow^!_{simp} (S', \theta_{simp})$
Build constraint graph
$(G(S'), \theta_{simp}) \Longrightarrow^!_{cyc} (G, \theta_{cyc})$
Constraint resolution
$(G, \theta_{cyc}) \Longrightarrow^!_{sol} (G', \theta_{sol})$
Unification
$(G'[TV(G')], \theta_{sol}) \Longrightarrow^!_{unif} ((\emptyset, \emptyset), \theta)$
Coercion generation and insertion
$\theta\Gamma \vdash \theta t \rightsquigarrow u : \theta\tau$
Output
term $u$, type $\theta\tau$, context $\theta\Gamma$

**Fig. 5.** Top-level control flow of the subtyping algorithm

### 3.2 Constraint generation

The algorithm for constraint generation is described by a judgement $\Gamma \vdash t : \tau \rhd S$ defined by the rules shown in Figure 6. Given a term $t$ and a context $\Gamma$, the algorithm returns a type $\tau$, as well as a set of *equality* and *subtype constraints* $S$ denoted by infix "$\doteq$" and "<:", respectively. The equality constraints are solved using unification, whereas the subtype constraints are simplified to atomic constraints and then solved using the graph-based algorithm mentioned above. The only place where new constraints are generated is the rule SubCT-App for function applications $t_1\ t_2$. It generates an equality constraint ensuring that the type of $t_1$ is actually a function type, as well as a subtype constraint ensuring that the type of $t_2$ is a subtype of the argument type of $t_1$.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rhd \emptyset}\ \textsc{SubCT-Var} \qquad \frac{\Sigma(c) = \sigma}{\Gamma \vdash c_{[\overline{\alpha} \mapsto \overline{\tau}]} : \sigma[\overline{\alpha} \mapsto \overline{\tau}] \rhd \emptyset}\ \textsc{SubCT-Const}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma \rhd S}{\Gamma \vdash \lambda x : \tau.\ t : \tau \to \sigma \rhd S}\ \textsc{SubCT-Abs}$$

$$\frac{\Gamma \vdash t_1 : \tau \rhd S_1 \qquad \Gamma \vdash t_2 : \sigma \rhd S_2 \qquad \alpha, \beta \text{ fresh}}{\Gamma \vdash t_1\ t_2 : \beta \rhd S_1 \cup S_2 \cup \{\tau \doteq \alpha \to \beta, \sigma <: \alpha\}}\ \textsc{SubCT-App}$$

**Fig. 6.** Constraint generation rules

Note that as a first step not shown here, the type-free term input by the user is augmented with type variables: $\lambda x.\ t$ becomes $\lambda x : \beta.\ t$ and $c$ becomes $c_{[\overline{\alpha} \mapsto \overline{\beta}]}$, where all the $\beta$s must be distinct and new.

## 3.3 Constraint simplification

The constraints generated in the previous step are now simplified by repeatedly applying the transformation rules shown in Figure 7. The states that the transformation operates on are pairs whose first component contains the current set of constraints, while the second component is used to accumulate the substitutions computed during the transformation. As a starting state of the transformation, we use the pair $(S, \emptyset)$. The rule Decompose splits up inequations between complex types into simpler inequations or equations according to the *variance* of the outermost type constructor. For this purpose, we introduce a *variance operator*, which is defined as follows.

**Definition 2 (Variance operator).** *Let $map_C$ be a map function for the type constructor $C$ of arity $n$ in the set $\mathcal{M}$. We use the abbreviation*

$$var_C^i(\tau, \sigma) = \begin{cases} \tau <: \sigma & \textit{if } C \textit{ is covariant in the } i\textit{-th argument wrt. } map_C \\ \sigma <: \tau & \textit{if } C \textit{ is contravariant in the } i\textit{-th argument wrt. } map_C \end{cases}$$

*for $1 \leq i \leq n$. If there is no such $map_C$, then we define for $1 \leq i \leq n$:*

$$var_C^i(\tau, \sigma) = \tau \doteq \sigma.$$

Thus, if no map function is associated with a particular type constructor, it is considered to be *invariant*, causing the algorithm to generate equations instead of inequations. Equations are dealt with by rule UNIFY using ordinary unification. Since our subtyping relation is structural, an inequation having a type variable on one side, and a complex type on the other side can only be solved by instantiating the type variable with a type whose outermost type constructor equals that of the complex type on the other side. This is expressed by the two symmetric rules EXPAND-L and EXPAND-R. Finally, inequations with an atomic type on both sides are eliminated by rule ELIMINATE, provided they conform to the subtyping relation.

DECOMPOSE

$(\{C\ \tau_1\ \dots\ \tau_n <: C\ \sigma_1\ \dots\ \sigma_n\} \uplus S, \theta) \Longrightarrow_{simp} \left(\left\{var_C^i\,(\tau_i, \sigma_i) \mid i = 1 \dots n\right\} \cup S, \theta\right)$

UNIFY

$(\{\tau \doteq \sigma\} \uplus S, \theta) \Longrightarrow_{simp} (\theta' S, \theta' \circ \theta)$

where $\theta' = mgu(\tau, \sigma)$

EXPAND-L

$(\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \uplus S, \theta) \Longrightarrow_{simp} (\theta'\,(\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \cup S)\,, \theta' \circ \theta)$

where $\theta' = \{\alpha \mapsto C\ \alpha_1\ \dots\ \alpha_n\}$
and $\alpha_1\ \dots\ \alpha_n$ are fresh variables

EXPAND-R

$(\{C\ \tau_1\ \dots\ \tau_n <: \alpha\} \uplus S, \theta) \Longrightarrow_{simp} (\theta'\,(\{C\ \tau_1\ \dots\ \tau_n <: \alpha\} \cup S)\,, \theta' \circ \theta)$

where $\theta' = \{\alpha \mapsto C\ \alpha_1\ \dots\ \alpha_n\}$
and $\alpha_1\ \dots\ \alpha_n$ are fresh variables

ELIMINATE

$(\{U <: T\} \uplus S, \theta) \Longrightarrow_{simp} (S, \theta)$

where $U, T$ are base types
and $U <: T$

**Fig. 7.** Rule-based constraint simplification $\Longrightarrow_{simp}$

We apply these rules repeatedly to the constraint set until none of the rules is applicable. Therefore, we use the notation $\Longrightarrow_{simp}^{!}$.

**Definition 3.** *(Normal form) For a relation $\Longrightarrow$ we write*

$$X \Longrightarrow^{!} X'$$

*if $X \Longrightarrow^{*} X'$ and $X'$ is in* normal form *wrt. $\Longrightarrow$.*

**Definition 4 (Atomic constraint).** *We call a subtype constraint* atomic *if it corresponds to one of the following constraints ($\alpha, \beta$ are type variables, $T$ is a base type):*

$$\alpha <: \beta \qquad \alpha <: T \qquad T <: \alpha$$

If none of the rules is applicable, the algorithm terminates in a state $(S', \theta_{simp})$, where $S'$ either consists only of atomic constraints, or $S'$ contains an inequation $C_1\ \overline{\tau} <: C_2\ \overline{\sigma}$ with $C_1 \neq C_2$ or an equation $\tau \doteq \sigma$ such that $\tau$ and $\sigma$ are not unifiable. In the latter two cases, the type inference algorithm fails.

An interesting question is whether such a state or a failure is always reached after a finite number of iterations. It is obvious that the simplification of the constraint $\alpha <: C\ \alpha$ will never terminate. Bourdoncle and Merz [2] have pointed out that checking whether the initial constraint set has a *weak unifier* is sufficient to avoid nontermination. We provide a proof for this statement in §5. Weak unification differs from standard unification in that it identifies base types, which is necessary since two types $\tau$ and $\sigma$ with $\tau <: \sigma$ need to be equal up to their base types.

**Definition 5 (Weak unification).** *A set of constraints $S$ is called* weakly unifiable *if there exists a substitution $\theta$ such that $\lceil \theta\tau \rceil = \lceil \theta\sigma \rceil$ for all $\tau <: \sigma \in S$, and $\theta\tau = \theta\sigma$ for all $\tau \doteq \sigma \in S$, where*

$$\begin{array}{lcl} \lceil \alpha \rceil & = & \alpha \\ \lceil T \rceil & = & T_0 \\ \lceil C\ \tau_1\ \dots\ \tau_n \rceil & = & C\ \lceil \tau_1 \rceil\ \dots\ \lceil \tau_n \rceil \end{array}$$

*and $T_0$ is a fixed base type not used elsewhere. In this case, $\theta$ is called a* weak unifier *of $S$, and $mgu\lceil S \rceil$ is called* weak most general unifier *of $S$.*

Weak unification is merely used as a termination-test in our algorithm before constraint simplification (see Figure 5).

## 3.4 Solving subtype constraints on a graph

An efficient and logically clean way to reason about atomic subtype constraints is to represent the types as nodes of a directed graph with arcs given by the constraints themselves. Concretely, this means that a subtype constraint $\sigma <: \tau$ is represented by the arc $(\sigma, \tau)$. This allows us to speak of predecessors and successors of a type.

**Definition 6 (Constraint graph).** *For a constraint set $S$, we denote by*

$$G(S) = (\bigcup \{\{\tau, \sigma\} \mid \tau <: \sigma \in S\}, \{(\tau, \sigma) \mid \tau <: \sigma \in S\})$$

*the* constraint graph *corresponding to $S$.*

Given a graph $G = (V, E)$, the subgraph induced by a vertex set $X \subseteq V$ is denoted by $G[X] = (X, (X \times X) \cap E)$. The set of type variables contained in the vertex set of $G$ is denoted by $TV(G)$.

In what follows, we write $\sigma \prec: \tau$ for the subtyping relation on base types induced by the set of coercions $\mathcal{C}$, which is defined by

$$\prec: \ = \ \{(T, U) \mid c : T \to U \in \mathcal{C}\}^*$$

**Definition 7 (Solution).** *A type substitution $\theta$ is a* solution *of the constraint set $S$ iff for all $\tau \doteq \sigma \in S$ it holds $\theta\tau = \theta\sigma$ and for all $\tau <: \sigma \in S$ there exists a coercion $c$ such that $\theta\tau <:_c \theta\sigma$ holds.*

*Moreover, we say that a type substitution $\theta$ is a* solution *of the constraint graph $G = (V, E)$ iff for all $(\sigma, \tau) \in E$ it holds that $\theta\sigma \prec: \theta\tau$.*

**Lemma 1 (Equivalence of the solution notions).** *$\theta$ is a solution of the atomic constraint set $S$ iff $\theta$ is a solution of $G(S)$.*

*Proof.* $S$ is an atomic constraint set. Hence, it does not contain unification constraints. By the definition of $G(S)$ the edges of $G(S)$ corresponding exactly to the subtype constraints in $S$. Thus, both notions of a solving substitution are equivalent. □

**Graph construction** Building such a constraint graph is straightforward. We only need to watch out for cycles. Since the subtype relation is a partial order and therefore antisymmetric, at most one base type should occur in a cycle. In other words, if the elements of the cycle are not unifiable, the inference will fail. Unifiable cycles should be eliminated with the iterated application of the rule Cycle-Elim shown in Figure 8.

$$
\begin{array}{lll}
\textsc{Cycle-Elim} & & \\
((V,E),\theta) & \Longrightarrow_{cyc} & ((V \setminus K \cup \{\tau_K\}, E' \cup P \times \{\tau_K\} \cup \{\tau_K\} \times S), \theta_K \circ \theta) \\
& & \text{where } K \text{ is a cycle in } (V,E) \\
& & \text{and } \theta_K = mgu(K) \\
& & \text{and } \{\tau_K\} = \theta_K K \\
& & \text{and } E' = \{(\tau,\sigma) \in E \mid \tau \notin K, \sigma \notin K\} \\
& & \text{and } P = \{\tau \mid \exists \sigma \in K.\ (\tau,\sigma) \in E\} \setminus K \\
& & \text{and } S = \{\tau \mid \exists \sigma \in K.\ (\tau,\sigma) \in E\} \setminus K
\end{array}
$$

**Fig. 8.** Rule-based cycle elimination $\Longrightarrow_{cyc}$

Figure 9 visualizes an example of cycle elimination. We call the substitution obtained from cycle elimination $\theta_{cyc}$.

**Constraint resolution** Now we must find an assignment for all variables that appear in the graph $G = (V, E)$. We use an algorithm that is based on the approach presented in [19]. First, we define some basic lattice-theoretic notions.

**Definition 8.** *Let $S, T, T'$ denote base types and $X$ a set of base types. With respect to the given subtype relation $\prec:$ we define:*

- $\overline{T} = \{T' \mid T \prec: T'\}$*, the set of* supertypes
- $\underline{T} = \{T' \mid T' \prec: T\}$*, the set of* subtypes
- $T \sqcup S \in \overline{T} \cap \overline{S}$ *and* $\forall U \in \overline{T} \cap \overline{S}.\ T \sqcup S \prec: U$*, the* supremum *of $S$ and $T$*

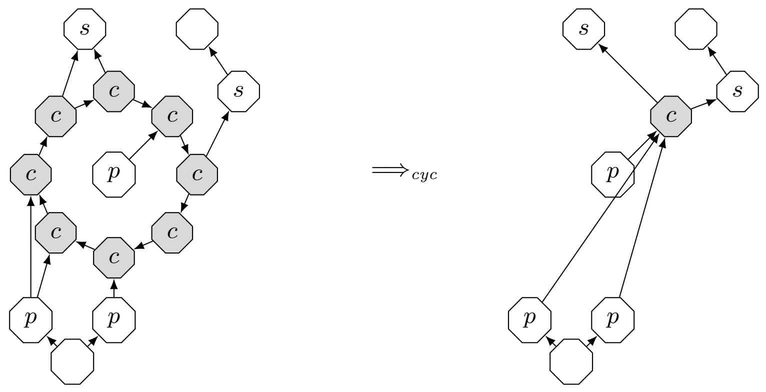


**Fig. 9.** Collapse of a cycle in a graph

- $T \sqcap S \in \underline{T} \cap \underline{S}$ *and* $\forall L \in \underline{T} \cap \underline{S}.\ L \prec: T \sqcap S$, *the* infimum *of* $S$ *and* $T$
- $\bigsqcup X \in \bigcap_{T \in X} \overline{T}$ *and* $\forall U \in \bigcap_{T \in X} \overline{T}.\ \bigsqcup X \prec: U$, *the* supremum *of* $X$
- $\bigsqcap X \in \bigcap_{T \in X} \underline{T}$ *and* $\forall L \in \bigcap_{T \in X} \underline{T}.\ L \prec: \bigsqcap X$, *the* infimum *of* $X$.

*Note that, depending on $\prec:$, suprema or infima may not exist.*

*Given a type variable $\alpha$ in the constraint graph $G = (V, E)$, we define:*

- $P_\alpha = \{T \mid (T, \alpha) \in E^+\}$, *the set of all base type* predecessors *of* $\alpha$
- $S_\alpha = \{T \mid (\alpha, T) \in E^+\}$, *the set of all base type* successors *of* $\alpha$.

*$E^+$ is the transitive closure of the edges of $G$.*

The algorithm assigns base types to type variables that have base type successors or predecessors until no such variables are left using the rules shown in Figure 10. The resulting substitution is referred to as $\theta_{sol}$.

The original algorithm described by Wand and O'Keefe [19] is designed to be complete for subtype relations that form a tree. It only uses the rules Assign-Inf and Fail-Inf without the check if $S_\alpha$ is empty. It assigns each type variable $\alpha$ the infimum $\bigsqcap S_\alpha$ of its upper bounds, and then checks whether the assigned type is greater than all lower bounds $P_\alpha$. If $\bigsqcap S_\alpha$ does not exist, their algorithm fails. If $S_\alpha$ is empty, its infimum only exists if there is a greatest type, which exists in a tree but not in a forest. In order to avoid this failure in the absence of a greatest type, our algorithm does not compute the infimum/supremum of the empty set, and is symmetric in successors/predecessors.

After constraint resolution, unassigned variables can only occur in the resulting graph in weakly connected components that do not contain any base types. All variables in a single weakly connected component should be unified. This is done by the rule Unify-WCC shown in Figure 11 and produces the final substitution $\theta$.

*Example 2.* Going back to Example 1, we apply our algorithm to the term $leq_{[\alpha \mapsto \alpha_3]}\ n\ i$. Figure 12 shows the derivation of the initial constraint set. Simplifying the generated

Assign-Sup

$$(G,\theta) \Longrightarrow_{sol} (\{\alpha \mapsto \bigsqcup P_\alpha\}G, \{\alpha \mapsto \bigsqcup P_\alpha\} \circ \theta)$$
$$\text{if } \alpha \in TV(G) \wedge P_\alpha \neq \emptyset \wedge \exists \bigsqcup P_\alpha \wedge \forall T \in S_\alpha.\ \bigsqcup P_\alpha \prec\colon T$$

Fail-Sup

$$(G,\theta) \Longrightarrow_{sol} \text{FAIL}$$
$$\text{if } \alpha \in TV(G) \wedge P_\alpha \neq \emptyset \wedge (\nexists \bigsqcup P_\alpha \vee \exists T \in S_\alpha.\ \bigsqcup P_\alpha \not\prec\colon T)$$

Assign-Inf

$$(G,\theta) \Longrightarrow_{sol} (\{\alpha \mapsto \bigsqcap S_\alpha\}G, \{\alpha \mapsto \bigsqcap S_\alpha\} \circ \theta)$$
$$\text{if } \alpha \in TV(G) \wedge S_\alpha \neq \emptyset \wedge \exists \bigsqcap S_\alpha \wedge \forall T \in P_\alpha.\ T \prec\colon \bigsqcap S_\alpha$$

Fail-Inf

$$(G,\theta) \Longrightarrow_{sol} \text{FAIL}$$
$$\text{if } \alpha \in TV(G) \wedge S_\alpha \neq \emptyset \wedge (\nexists \bigsqcap S_\alpha \vee \exists T \in P_\alpha.\ T \not\prec\colon \bigsqcap S_\alpha)$$

**Fig. 10.** Rule-based constraint resolution $\Longrightarrow_{sol}$

Unify-WCC

$$(G,\theta) \Longrightarrow_{unif} (G[V \setminus W], mgu(W) \circ \theta)$$
$$\text{where } W \text{ is a weakly connected component of } G = (V,E)$$

**Fig. 11.** Rule-based WCC-unification $\Longrightarrow_{unif}$

constraints yields the substitution $\theta_{simp} = \{\alpha_1 \mapsto \alpha_3, \alpha_2 \mapsto \alpha_3, \beta_1 \mapsto \mathbb{B}, \beta_2 \mapsto \alpha_3 \to \mathbb{B}\}$ and the atomic constraint set $\{\mathbb{N} <: \alpha_2, \mathbb{Z} <: \alpha_1\}$. This yields the constraint graph shown in Figure 13. The constraint resolution algorithm assigns $\alpha_3$ the least upper bound of $\{\mathbb{N}, \mathbb{Z}\}$, which is $\mathbb{Z}$. The resulting substitution is $\theta_{sol} = \{\alpha_1 \mapsto \mathbb{Z}, \alpha_2 \mapsto \mathbb{Z}, \alpha_3 \mapsto \mathbb{Z}, \beta_1 \mapsto \mathbb{B}, \beta_2 \mapsto \mathbb{Z} \to \mathbb{B}\}$. Since there are no unassigned variables in the remaining constraint graph, Unify-WCC is inapplicable and $\theta$, the final result, is $\theta_{sol}$.

$$\dfrac{\dfrac{\dfrac{\Sigma(leq) = \alpha \to \alpha \to \mathbb{B}}{\Gamma \vdash leq_{[\alpha \mapsto \alpha_3]} : \alpha_3 \to \alpha_3 \to \mathbb{B} \rhd \emptyset} \quad \dfrac{\Sigma(n) = \mathbb{N}}{\Gamma \vdash n : \mathbb{N} \rhd \emptyset} \quad \alpha_2, \beta_2 \text{ fresh}}{\Gamma \vdash leq_{[\alpha \mapsto \alpha_3]}\ n : \beta_2 \rhd \{\alpha_3 \to \alpha_3 \to \mathbb{B} \doteq \alpha_2 \to \beta_2, \mathbb{N} <: \alpha_2\}} \quad \dfrac{\Sigma(i) = \mathbb{Z}}{\Gamma \vdash i : \mathbb{Z} \rhd \emptyset} \quad \alpha_1, \beta_1 \text{ fresh}}{\Gamma \vdash leq_{[\alpha \mapsto \alpha_3]}\ n\ i : \beta_1 \rhd \{\alpha_3 \to \alpha_3 \to \mathbb{B} \doteq \alpha_2 \to \beta_2, \beta_2 \doteq \alpha_1 \to \beta_1, \mathbb{N} <: \alpha_2, \mathbb{Z} <: \alpha_1\}}$$

**Fig. 12.** Example for constraint generation

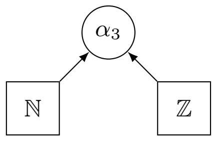


**Fig. 13.** Constraint graph of $leq_{[\alpha \mapsto \alpha_3]}\ n\ i$

In §6 we will see that the constraint resolution algorithm defined in this subsection is not complete in general but is complete if the partial order on base types is a disjoint union of lattices.

### 3.5 Coercion insertion

Finally, we have a solving substitution $\theta$. Applying this substitution to the initial term will produce a term that can always be coerced to a type correct term by means of the coercion insertion judgement shown in Figure 3. We inspect this correctness statement and the termination of our algorithm in §5.

## 4 Interaction with type classes

### 4.1 Type classes in Isabelle

The type system of the Isabelle proof assistant is a simply typed lambda calculus with Hindley-Milner polymorphism and type classes. Our extension of type inference with subtyping interacts in a rather subtle way with the type class system of Isabelle. The type class system is described in [10] and [20]. For our purpose it suffices to abstract the system to the following facts.

Type classes represent sets of types. The intersection of finitely many type classes is called a sort. Sorts are quasi-ordered (we call the relation "$\sqsubseteq$") and a maximal sort $\top$ exists. $\top$ contains all types. All type variables are annotated with sorts. We use the notation $\alpha :: \mathcal{S}$ for this. A type $T$ can be tested for membership in a sort $\mathcal{S}$ via the judgement $\vdash_{sort} T : \mathcal{S}$. Further, a function $arity(C, \mathcal{S})$ is given. *arity* takes a type constructor $C$ of arity $n$ and a sort $\mathcal{S}$ as input and returns a tuple of $n$ sorts $(\mathcal{S}_1, \ldots, \mathcal{S}_n)$. The interpretation of this function is that $\vdash_{sort} C\ \tau_1 \ldots \tau_n : \mathcal{S}$ holds only if $\vdash_{sort} \tau_i : \mathcal{S}_i$ holds $\forall i = 1 \ldots n$. $arity(C, \mathcal{S})$ can fail if $C$ constructs types that do not belong to sort $\mathcal{S}$. Also, a unification algorithm that respects the type classes is required[11]. For example, $\alpha :: \mathcal{S}$ and $\mathbb{X}$ are not unifiable if $\vdash_{sort} \mathbb{X} : \mathcal{S}$ does not hold and the unifier of $\alpha :: \mathcal{S}$ and $\beta :: \mathcal{T}$ is a type variable annotated with the intersection of the sorts $\mathcal{S}$ and $\mathcal{T}$.

### 4.2 Interaction with constraint preprocessing

The constraint generation rules are not dramatically affected by type classes. We only need to pass the variable annotations around. Fresh variables get annotated with $\top$.

As we use the weak unification test only to ensure the termination of constraint simplification we will not need to check sort consistency at this point. So, exactly the same test on subtype constraints as in §3 is enough.

During constraint simplification we need to ensure correct sort annotation when we replace a variable with a constructed type. Therefore, we update the simplification rules Expand-L and Expand-R as shown in Figure 14.

Expand-L

$$(\{\alpha :: \mathcal{S} <: C\ \tau_1\ \ldots\ \tau_n\} \uplus S, \theta) \Longrightarrow_{simp} (\theta'(\{\alpha :: \mathcal{S} <: C\ \tau_1\ \ldots\ \tau_n\} \cup S), \theta' \circ \theta)$$

where $\theta' = \{\alpha \mapsto C\ (\alpha_1 :: \mathcal{S}_1)\ \ldots\ (\alpha_n :: \mathcal{S}_n)\}$
and $\mathit{arity}(C, \mathcal{S}) = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$
and $\alpha_1\ \ldots\ \alpha_n$ are fresh variables

Expand-R

$$(\{C\ \tau_1\ \ldots\ \tau_n <: \alpha :: \mathcal{S}\} \uplus S, \theta) \Longrightarrow_{simp} (\theta'(\{C\ \tau_1\ \ldots\ \tau_n <: \alpha :: \mathcal{S}\} \cup S), \theta' \circ \theta)$$

where $\theta' = \{\alpha \mapsto C\ (\alpha_1 :: \mathcal{S}_1)\ \ldots\ (\alpha_n :: \mathcal{S}_n)\}$
and $\mathit{arity}(C, \mathcal{S}) = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$
and $\alpha_1\ \ldots\ \alpha_n$ are fresh variables

**Fig. 14.** Rule-based constraint simplification, extended to handle type classes

## 4.3 Interaction with constraint resolution

Assuming a new unification function that respects sort annotations, the process of building the constraint graph and eliminating cycles works the same way as before. Constraint resolution is more problematic. Here, every assignment must be "sort-correct".

*Example 3.* Consider the following two examples in Figure 15. In example (a) our algorithm would try to assign the type $\mathbb{N}$ to $\alpha :: \mathit{Field}$. This, however, is wrong since $\mathbb{N}$ does not belong to the sort *Field*. Instead, the "smallest" supertype of $\mathbb{N}$ that is of sort *Field* should be assigned to $\alpha$ . In our case this is $\mathbb{R}$. Example (b) demonstrates another kind of influence of sorts on the assignment of variables. In this case $\alpha$ is annotated with $\top$ which would not contradict the assignment of $\mathbb{N}$ to $\alpha$. Still this assignment is wrong because $\mathbb{N}$ does not have any subtypes that belong to the sort *Field* which is required for the assignment of $\beta :: \mathit{Field}$. To get the right assignment we need to find a supertype of $\mathbb{N}$ that has a subtype of sort *Field*. Again the solution is $\mathbb{R}$.

We integrate the observations in our algorithm by defining a new supremum and infimum that depend on the sort of the variable to be assigned and a set of sorts belonging to the predecessors/successors of this variable.

**Definition 9.** *Let $S, T, T', T''$ denote base types, $X$ a set of base types, $\mathcal{S}$ a sort and $\mathcal{X}$ a set of type variables. We define wrt. the given subtype relation $\prec$::*

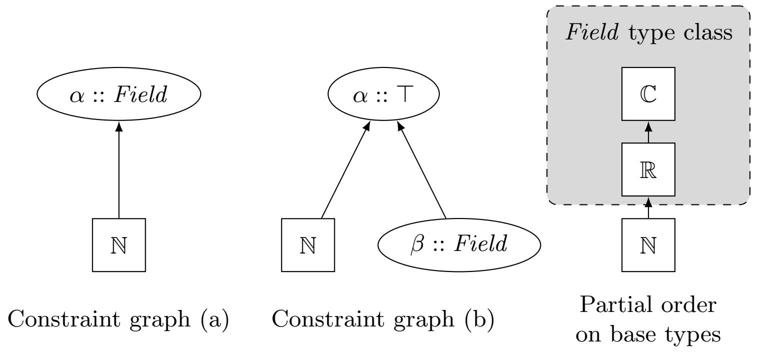


**Fig. 15.** Examples for type class-subtype interference

- $\overline{T}^{\mathcal{X}}_{\mathcal{S}} = \{T' \mid T \prec: T', \vdash_{sort} T' : \mathcal{S},$
  $\forall \beta :: \mathcal{T} \in \mathcal{X}.\ \exists T''. \left(\forall U \in P_\beta.\ U \prec: T''\right), T'' \prec: T', \vdash_{sort} T'' : \mathcal{T}\}$, *the set of* supertypes *that are of sort $\mathcal{S}$ and have valid subtypes for every type variable in $\mathcal{X}$*
- $\underline{T}^{\mathcal{X}}_{\mathcal{S}} = \{T' \mid T' \prec: T, \vdash_{sort} T' : \mathcal{S},$
  $\forall \beta :: \mathcal{T} \in \mathcal{X}.\ \exists T''.\ \left(\forall U \in S_\beta.\ T'' \prec: U\right), T' \prec: T'', \vdash_{sort} T'' : \mathcal{T}\}$, *the set of* subtypes *that are of sort $\mathcal{S}$ and have valid supertypes for every type variable in $\mathcal{X}$*
- $T \sqcup^{\mathcal{X}}_{\mathcal{S}} S \in \overline{T}^{\mathcal{X}}_{\mathcal{S}} \cap \overline{S}^{\mathcal{X}}_{\mathcal{S}}$ *and* $\forall U \in \overline{T}^{\mathcal{X}}_{\mathcal{S}} \cap \overline{S}^{\mathcal{X}}_{\mathcal{S}}.\ T \sqcup^{\mathcal{X}}_{\mathcal{S}} S \prec: U$, *the* supremum *of $S$ and $T$*
- $T \sqcap^{\mathcal{X}}_{\mathcal{S}} S \in \underline{T}^{\mathcal{X}}_{\mathcal{S}} \cap \underline{S}^{\mathcal{X}}_{\mathcal{S}}$ *and* $\forall L \in \underline{T}^{\mathcal{X}}_{\mathcal{S}} \cap \underline{S}^{\mathcal{X}}_{\mathcal{S}}.\ L \prec: T \sqcap^{\mathcal{X}}_{\mathcal{S}} S$ *the* infimum *of $S$ and $T$*
- $\bigsqcup^{\mathcal{X}}_{\mathcal{S}} X \in \bigcap_{T \in X} \overline{T}^{\mathcal{X}}_{\mathcal{S}}$ *and* $\forall U \in \bigcap_{T \in X} \overline{T}^{\mathcal{X}}_{\mathcal{S}}.\ \bigsqcup^{\mathcal{X}}_{\mathcal{S}} X \prec: U$, *the* supremum *of $X$*
- $\bigsqcap^{\mathcal{X}}_{\mathcal{S}} X \in \bigcap_{T \in X} \underline{T}^{\mathcal{X}}_{\mathcal{S}}$ *and* $\forall L \in \bigcap_{T \in X} \underline{T}^{\mathcal{X}}_{\mathcal{S}}.\ L \prec: \bigsqcap^{\mathcal{X}}_{\mathcal{S}} X$, *the* infimum *of $X$.*

*Further let $\alpha$ be a type variable in the constraint graph $G = (V, E)$. We denote the* transitive closure *of edges of $G$ by $E^+$ and define:*

- $P_\alpha = \{T \mid (T, \alpha :: \mathcal{S}) \in E^+, T$ *base type*$\}$
  *the set of all base type* predecessors *of $\alpha :: \mathcal{S}$*
- $S_\alpha = \{T \mid (\alpha :: \mathcal{S}, T) \in E^+, T$ *base type*$\}$
  *the set of all base type* successors *of $\alpha :: \mathcal{S}$*
- $\mathcal{P}_\alpha = \{\beta :: \mathcal{T} \mid (\beta :: \mathcal{T}, \alpha :: \mathcal{S}) \in E^+\}$
  *the set of all* type variable predecessors *of $\alpha :: \mathcal{S}$*
- $\mathcal{S}_\alpha = \{\beta :: \mathcal{T} \mid (\alpha :: \mathcal{S}, \beta :: \mathcal{T}) \in E^+\}$
  *the set of all* type variable successors *of $\alpha :: \mathcal{S}$.*

Figure 16 shows the constraint resolution rules updated according to the new notion of the supremum/infimum.

Assign-Sup

$$(G,\theta) \Longrightarrow_{sol} \Big(\{\alpha \mapsto \textstyle\bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha\}G, \{\alpha \mapsto \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha\} \circ \theta\Big)$$
$$\text{if } \alpha :: \mathcal{S} \in TV(G) \wedge P_\alpha \neq \emptyset \wedge \exists \textstyle\bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha \wedge \forall T \in S_\alpha .\ \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha \prec: T$$

Fail-Sup

$$(G,\theta) \Longrightarrow_{sol} \text{FAIL}$$
$$\text{if } \alpha :: \mathcal{S} \in TV(G) \wedge P_\alpha \neq \emptyset \wedge (\nexists \textstyle\bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha \vee \exists T \in S_\alpha .\ \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha \not\prec: T)$$

Assign-Inf

$$(G,\theta) \Longrightarrow_{sol} \Big(\{\alpha \mapsto \textstyle\bigsqcap_{\mathcal{S}}^{\mathcal{S}_\alpha} S_\alpha\}G, \{\alpha \mapsto \bigsqcap_{\mathcal{S}}^{\mathcal{S}_\alpha} S_\alpha\} \circ \theta\Big)$$
$$\text{if } \alpha :: \mathcal{S} \in TV(G) \wedge S_\alpha \neq \emptyset \wedge \exists \textstyle\bigsqcap_{\mathcal{S}}^{\mathcal{S}_\alpha} S_\alpha \wedge \forall T \in P_\alpha .\ T \prec: \bigsqcap_{\mathcal{S}}^{\mathcal{S}_\alpha} S_\alpha$$

Fail-Inf

$$(G,\theta) \Longrightarrow_{sol} \text{FAIL}$$
$$\text{if } \alpha :: \mathcal{S} \in TV(G) \wedge S_\alpha \neq \emptyset \wedge (\nexists \textstyle\bigsqcap_{\mathcal{S}}^{\mathcal{S}_\alpha} S_\alpha \vee \exists T \in P_\alpha .\ T \not\prec: \bigsqcap_{\mathcal{S}}^{\mathcal{S}_\alpha} S_\alpha)$$

**Fig. 16.** Rule-based constraint resolution (type classes) $\Longrightarrow_{sol}$

After integrating those extensions, the algorithm still always terminates and is sound. However, completeness can not be reached even with very strong restrictions of the base type poset, without further assumptions about the sort quasi-order. We discuss some examples for incompleteness in §6.

## 5 Total correctness

To prove total correctness, we need to show that for any input $t$ and $\Gamma$, the algorithm either returns a substitution $\theta$, a well-typed term $u$ together with its type $\theta\tau$ or indicates a failure. Failures may occur at any computation of a most general unifier, a sort arity, during the weak unification test, or explicitly at the reduction steps Fail-Sup and Fail-Inf in the constraint resolution phase. Below we discuss correctness and termination. Since the reduction rules in each phase are applied nondeterministically, the algorithm may output different substitutions for the same input term $t$ and context $\Gamma$. By $AlgSol(\Gamma, t)$ we denote the set of all such substitutions.

**Theorem 1 (Correctness).** *For a given term $t$ in the context $\Gamma$, assume $\theta \in AlgSol(\Gamma, t)$. Then there exist a term $u$ and a type $\tau$, such that $\theta\Gamma \vdash \theta t \leadsto u : \tau$ and $\theta\Gamma \vdash u : \tau$.*

Instead of working with the notion *AlgSol* and the whole algorithm as a black box, we reformulate the correctness theorem to give names to the intermediate results. Then, we can show correctness statements about each phase of the algorithm separately.

**Theorem 2 (Correctness).** *Suppose that a given term $t$ in the context $\Gamma$ passes the phases of the presented algorithm without failure:*

$$\Gamma \vdash t : \tau \rhd S, \tag{1}$$
$$(S, \emptyset) \quad \Longrightarrow^{!}_{simp} \quad (S', \theta_{simp}), \tag{2}$$
$$(G(S'), \theta_{simp}) \quad \Longrightarrow^{!}_{cyc} \quad (G, \theta_{cyc}), \tag{3}$$
$$(G, \theta_{cyc}) \quad \Longrightarrow^{!}_{sol} \quad (G', \theta_{sol}), \tag{4}$$
$$(G'[TV(G')], \theta_{sol}) \quad \Longrightarrow^{!}_{unif} \quad ((\emptyset, \emptyset), \theta). \tag{5}$$

*Then there exists a term $u$, such that*

$$\theta\Gamma \vdash \theta t \rightsquigarrow u : \theta\tau, \tag{6}$$
$$\theta\Gamma \vdash u : \theta\tau. \tag{7}$$

*Proof.* In Lemma 3, we prove that (7) follows from (6).

In order to prove (6), we observe that any solution of the constraint set $S$ produced in (1), has the needed property. This is shown in Lemma 4. Thus, we need to show that $\theta$ is a solution of $S$. We achieve this by proving that each reduction phase results in a substitution that solves a certain part of the restrictions induced by the constraint set. More precisely, there exists a substitution $\delta_4$ such that $\theta = \delta_4 \circ \theta_{sol}$ and $\delta_4$ solves $G'[TV(G')]$ provided (5), which is shown in Lemma 8. Clearly, $\delta_4$ also solves $G'$, since in $G'$ there are no edges between simple types and a type variables.

Further, in Lemma 7 we show that (4) implies the existence of a $\delta_3$ such that $\theta_{sol} = \delta_3 \circ \theta_{cyc}$ and $\delta_3$ extends a solution of $G'$ to a solution of $G$, Lemma 6 and Lemma 1 show that there exists a substitution $\delta_2$ such that $\theta_{cyc} = \delta_2 \circ \theta_{simp}$ and $\delta_2$ extends a solution of $G$ to a solution of $S'$ provided (3), and finally Lemma 5 shows that $\theta_{simp}$ extends a solution of $S'$ to a solution of $S$ provided (2). Concatenating these results implies the fact that $\theta = \delta_4 \circ \delta_3 \circ \delta_2 \circ \theta_{simp}$ solves $S$. □

**Lemma 2 (Correctness of coercion generation).** *If $\tau <:_c \sigma$ then $\emptyset \vdash c : \tau \to \sigma$.*

*Proof.* A straightforward induction on the derivation of the coercion generation $\tau <:_c \sigma$. □

**Lemma 3 (Correctness of coercion insertion).** *If $\Gamma \vdash t \rightsquigarrow u : \tau$ then $\Gamma \vdash u : \tau$.*

*Proof.* A straightforward induction on the derivation of the coercion insertion $\Gamma \vdash t \rightsquigarrow u : \tau$ additionally using Lemma 2. □

**Lemma 4 (Non-failure of coercion insertion).** *Given a term $t$ and context $\Gamma$, if $\Gamma \vdash t : \tau \rhd S$ and $\theta$ is a solution of the constraint set $S$, then there exists a term $u$ such that $\theta\Gamma \vdash \theta t \rightsquigarrow u : \theta\tau$.*

*Proof.* A straightforward induction on the derivation of the constraint generation $\Gamma \vdash t : \tau \rhd S$. □

**Lemma 5 (Correctness of $\Longrightarrow^!_{simp}$).** *Assume $(S, \theta) \Longrightarrow^!_{simp} (S', \theta')$. Let $\theta''$ be a substitution that solves $S'$. Then there exists a substitution $\delta$ such that $\theta' = \delta \circ \theta$ and $\theta'' \circ \delta$ solves $S$.*

*Proof.* First, we prove a related property for a single application of a simplification rule, from which the above property follows by an induction on the number of simplification steps.

Assume $(S, \theta) \Longrightarrow_{simp} (S', \theta')$ and $\theta''$ is a solving substitution for $S'$. We show that in that case there exists a substitution $\delta$ such that $\theta' = \delta \circ \theta$ and $\theta'' \circ \delta$ is a solving substitution for $S$ by case distinction on the rules for $\Longrightarrow_{simp}$.

Decompose In this case $\theta' = \theta$ implies $\delta = \emptyset$. Therefore, we need to show that $\theta'' \circ \delta = \theta''$ is a solution for $\{C\ \tau_1\ \dots\ \tau_n <: C\ \sigma_1\ \dots\ \sigma_n\} \uplus S$. As we know, $\theta''$ solves the constraints $\left\{ var^i_C\,(\tau_i, \sigma_i) \mid i = 1 \dots n \right\} \cup S$.

If there is a known map function $m$ for $C$, then for all $i = 1 \dots n$ there exists $c_i$ such that either $\theta''\tau_i <:_{c_i} \theta''\sigma_i$ or $\theta''\tau_i <:_{c_i} \theta''\sigma_i$ holds according to the variance of the constructor $C$. Hence, $\theta''\,(C\ \tau_1\ \dots\ \tau_n) <:_{m\ c_1 \dots c_n} \theta''\,(C\ \sigma_1\ \dots\ \sigma_n)$ holds.

Otherwise, if no map function for $C$ is known, the fact that $\theta''$ solves $var^i_C\,(\tau_i, \sigma_i)$ implies $\theta''\tau_i = \theta''\sigma_i$ for all $i = 1 \dots n$. Hence, $\theta''\,(C\ \tau_1\ \dots\ \tau_n) <:_{id} \theta''\,(C\ \sigma_1\ \dots\ \sigma_n)$, because the constructed terms are equal after the application of $\theta''$.

Thus, $\theta''$ solves $\{C\ \tau_1\ \dots\ \tau_n <: C\ \sigma_1\ \dots\ \sigma_n\} \uplus S$.

Unify It is easy to see that $\delta = mgu\,\{\tau, \sigma\}$. Therefore, we have to prove that $\theta'' \circ \delta$ is a solving substitution for $\{\tau \doteq \sigma\} \uplus S$, provided that $\theta''$ solves $\delta S$. Then, there must be only already "solved" constraints in $\theta''(\delta S) = (\theta'' \circ \delta)S$. It follows that $\theta'' \circ \delta$ solves the constraints in $S$

Since $\delta$ is the most general unifier of $\tau$ and $\sigma$, it holds: $(\theta'' \circ \delta)\tau = \theta''(\delta\tau) = \theta''(\delta\sigma) = (\theta'' \circ \delta)\sigma$. Hence, $\theta'' \circ \delta$ also solves the constraint $\tau \doteq \sigma$.

Expand-L *and* Expand-R Both rules, Expand-L and Expand-R, behave in the same way.

We observe that $\delta = \{\alpha \mapsto C\ \alpha_1 \dots \alpha_n\}$[1] and $S' = \delta S$. Since $\theta''$ is a solving substitution for $S'$, $\theta''(\delta S) = (\theta'' \circ \delta)S$ contains only "solved" constraints. Thus, $\theta'' \circ \delta$ is solution for $S$.

Eliminate This step is eliminating already solved constraints. With $\delta = \emptyset$, $\theta'' \circ \delta = \theta''$ still solves $S$ together with the eliminated constraint $U <: T$ since $\theta''U = U$ and $\theta''T = T$. □

**Lemma 6 (Correctness of $\Longrightarrow^!_{cyc}$).** *Assume $(G, \theta) \Longrightarrow^!_{cyc} (G', \theta')$. Let $\theta''$ be a substitution that solves $G'$. Then there exists a substitution $\delta$ such that $\theta' = \delta \circ \theta$ and $\theta'' \circ \delta$ solves $G$.*

[1] or $\{\alpha \mapsto C\ (\alpha_1 :: \mathcal{S}_1)\ \dots\ (\alpha_n :: \mathcal{S}_n)\}$ in the case with type classes

*Proof.* The proof idea is the same as in Lemma 5 with a degenerate case distinction for the only rule.

Cycle-Elim Assuming that $\theta''$ is a solving substitution for $(V \setminus K \cup \{\tau_K\}, E' \cup P \times \{\tau_K\} \cup \{\tau_K\} \times S)$, we need to show that $\theta'' \circ \theta_K$ is a solution for $(V, E)$ with $\theta_K = mgu(K)$. That is, for all $(\tau, \sigma) \in E$ it must hold: $(\theta'' \circ \theta_K)\tau \prec: (\theta'' \circ \theta_K)\sigma$.

To prove this statement, we distinguish between four different edge categories. Note that it holds $\forall \tau \in K.\ \theta_K\tau = \tau_K$ and $\forall \tau \in V \setminus K.\ \theta_K\tau = \tau$ since there are only variables and base types in $V$.

1. $\tau \in V \setminus K, \sigma \in V \setminus K$. Then $(\tau, \sigma) \in E'$, such that:
$$(\theta'' \circ \theta_K)\tau = \theta''(\theta_K\tau) = \theta''\tau \prec: \theta''\sigma = \theta''(\theta_K\sigma) = (\theta'' \circ \theta_K)\sigma.$$
2. $\tau \in K, \sigma \in V \setminus K$. Then $(\tau, \sigma) \in \{\tau_K\} \times S$, such that:
$$(\theta'' \circ \theta_K)\tau = \theta''\tau_K \prec: \theta''\sigma = (\theta'' \circ \theta_K)\sigma.$$
3. $\tau \in V \setminus K, \sigma \in K$. Then $(\tau, \sigma) \in P \times \{\tau_K\}$, such that:
$$(\theta'' \circ \theta_K)\tau = \theta''\tau \prec: \theta''\tau_K = (\theta'' \circ \theta_K)\sigma.$$
4. $\tau \in K, \sigma \in K$. Then it holds:
$$(\theta'' \circ \theta_K)\tau = \theta''\tau_K = (\theta'' \circ \theta_K)\sigma.$$

□

**Lemma 7 (Correctness of $\Longrightarrow^!_{sol}$).** *Assume $(G, \theta) \Longrightarrow^!_{sol} (G', \theta')$. Let $\theta''$ be a substitution that solves $G'$. Then there exists a substitution $\delta$ such that $\theta' = \delta \circ \theta$ and $\theta'' \circ \delta$ solves $G$.*

*Proof.* Once again, the proof idea is the same as before. $\Longrightarrow_{sol}$ has two non-failing cases, but since they are symmetric to each other, we consider only Assign-Sup rule of the case with type classes.

Assign-Sup We write in the following, $\delta$ for the substitution $\{\alpha \mapsto \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha\}$. Assuming that $\theta''$ is a solving substitution for $\delta G$ and the given side conditions $\alpha :: \mathcal{S} \in TV(G)$, $P_\alpha \neq \emptyset$, $\exists \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha$, and $\forall T \in S_\alpha.\ \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha \prec: T$, we need to show that $\theta'' \circ \delta$ is a solving substitution for $G = (V, E)$. Therefore, we we distinguish between three categories for the edge $(\tau, \sigma) \in E$.

1. $\tau \in V \setminus \{\alpha\}, \sigma \in V \setminus \{\alpha\}$. Then $(\tau, \sigma) \in \delta G$, such that:
$$(\theta'' \circ \delta)\tau = \theta''\tau \prec: \theta''\sigma = (\theta'' \circ \delta)\sigma.$$
2. $\tau = \alpha, \sigma \in V \setminus \{\alpha\}$. Then, because of the side condition $\forall T \in S_\alpha.\ \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha \prec: T$ and $\theta''$ being a solving substitution, it holds:
$$(\theta'' \circ \delta)\tau = \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha \prec: \theta''\sigma = (\theta'' \circ \delta)\sigma.$$

3. $\tau \in V \setminus \{\alpha\}, \sigma = \alpha$. Then, because of the definition of the supremum of $P_\alpha$ and $\theta''$ being a solving substitution, it holds:

$$(\theta'' \circ \delta)\tau = \theta''\tau \prec: \bigsqcup_{\mathcal{S}}^{\mathcal{P}_\alpha} P_\alpha = (\theta'' \circ \delta)\sigma.$$

□

**Lemma 8 (Correctness of $\Longrightarrow^!_{unif}$).** *Assume $(G, \theta) \Longrightarrow^!_{unif} ((\emptyset, \emptyset), \theta')$. Then, there exists a substitution $\delta$ such that $\theta' = \delta \circ \theta$ and $\delta$ solves $G$.*

*Proof.* This proof uses again the very same technique as the previous ones. As in the proof of the correctness of $\Longrightarrow^!_{cyc}$, there is only one rule.

*Unify-WCC* Assuming that $\theta''$ solves $G[V \setminus W]$, we need to show that $\theta'' \circ mgu(W)$ solves $G = (V, E)$. Remembering that $W$ denotes a weakly connected component of $G$, we distinguish between two categories for the edge $(\tau, \sigma) \in E$. It holds $\forall \tau \in W.\ mgu(W)\tau =: \tau_W$ and $\forall \tau \in V \setminus W.\ mgu(W)\tau = \tau$.

1. $\tau \in V \setminus W, \sigma \in V \setminus W$. Then $(\tau, \sigma) \in G[V \setminus W]$, such that:

$$(\theta'' \circ mgu(W))\tau = \theta''\tau \prec: \theta''\sigma = (\theta'' \circ mgu(W))\sigma.$$

2. $\tau \in W, \sigma \in W$. Then it holds:

$$(\theta'' \circ mgu(W))\tau = \theta''\tau_W = (\theta'' \circ mgu(W))\sigma.$$

□

Thus, we know that if the algorithm terminates successfully, it returns a well-typed term. Moreover, it terminates for any input:

**Theorem 3 (Termination).** *The algorithm terminates for any input $t$ and $\Gamma$.*

*Proof.* The derivations using constraint generation and coercion generation and insertion typing rules clearly terminate since these rules are compositional, that is depending only on their subexpressions.

The termination of the weak unification test follows from the termination of the standard unification algorithm.

Less obvious is the termination of the repeated application of the reduction rules. We show that in the non-failing case the normal form is reached after a finite number of reduction steps by specifying a measure for each phase that decreases with every application of any rule belonging to this phase.

*Termination of $\Longrightarrow^!_{simp}$ assuming the weak unifiability of the original constraint set $S$ (Proof based on the proof of lemma 13 in [2])*

First of all, we show that after the application of any rule the resulting constraint set is weakly unifiable if the original constraint set was weakly unifiable. Let $S_i$ be the constraint set in step $i$, and $\theta_i$ be a weak unifier for $S_i$. If we apply rule Decompose or rule Eliminate, $\theta_i$ is still a weak substitution for the resulting constraint set. The application of rule Expand-L to the subtype constraint $\alpha <: C\ \tau_1\ \tau_2\ \dots\ \tau_n$ introduces new variables $\alpha_1, \dots, \alpha_n$ and applies the substitution $\{\alpha \mapsto C\ \alpha_1\ \alpha_2\ \dots\ \alpha_n\}$ to the whole constraint set. Since $\alpha <: C\ \tau_1\ \tau_2\ \dots\ \tau_n \in S_i$ and $\theta_i$ is a weak unifier of $S_i$, it follows that $\theta_i\alpha = C\ \sigma_1\ \dots\ \sigma_n$, where $\sigma_j = \lceil \theta_i \tau_j \rceil$ for all $1 \leq j \leq n$. Thus, $\theta_{i+1} = \{\alpha_1 \mapsto \sigma_1, \dots, \alpha_n \mapsto \sigma_n\} \circ \theta_i$ is a weak unifier of $S_{i+1}$. By a similar argument, it can also be shown that the rule Expand-R preserves weak unifiability. It is part of the algorithm to test whether the initial constraint set is weakly unifiable. We will only start simplifying the constraints if they pass the test. By induction, we obtain weak unifiability for every subtype constraint set produced by our simplification algorithm.

Now, we can define the decreasing measure. Let $\#(\tau)$ be the following recursively defined number:

$$\#(\tau) = \begin{cases} 1 & \text{if } \tau \text{ is a type variable or base type} \\ 1 + \sum\limits_{i=1}^{n} \#(\tau_i) & \text{if } \tau = C\ \tau_1\ \tau_2\ \dots\ \tau_n \end{cases}$$

Furthermore, we define two measures for a weakly unifiable subtype constraint set $S_i$:

$$|S_i| = \sum_{\tau \lhd \sigma \in S_i} \#(\tau) + \#(\sigma) \text{ and } |S_i|_{TV} = \sum_{x \in TV(S_i)} \#(\theta_i x)$$

where $\theta_i$ is a weak most general unifier for $S_i$, $TV(S_i)$ denotes the type variables occurring in $S_i$, and $\lhd$ stands for either $<:$ or $\doteq$.

Consider the lexicographic order on tuples $(|S_i|_{TV}, |S_i|)$ for the coercion sets $S_i$ in any step $i$. If rules Decompose or Eliminate are applied, then $|S_i|_{TV} = |S_{i+1}|_{TV}$ holds because $TV(S_i) = TV(S_{i+1})$ and $\theta_i = \theta_{i+1}$ obviously hold. Note that $\theta_i$ and $\theta_{i+1}$ (weak most general unifiers for $S_i$ and $S_{i+1}$) must exist, since $S_i$ and $S_{i+1}$ are weakly unifiable. On the other hand, $|S_i| = |S_{i+1}| + 2$ holds because the rules are either removing exactly two type constructors or exactly two base types.

If Expand-L is applied to the constraint $\alpha <: C\ \tau_1\ \tau_2\ \dots\ \tau_n$, we notice that the weak unifiers $\theta_i$ and $\theta_{i+1}$ only differ in the variables $\alpha$ and $\alpha_1, \dots, \alpha_n$. Furthermore, since $\theta_i$ and $\theta_{i+1}$ are weak most general unifiers, it holds that $\theta_i\alpha = \theta_{i+1}(C\ \alpha_1\ \alpha_2\ \dots\ \alpha_n)$. Now, we can conclude

$$\begin{aligned} \#(\theta_i\alpha) &= \#(\theta_{i+1}(C\ \alpha_1\ \alpha_2\ \dots\ \alpha_n)) \\ &= 1 + \sum_{j=1}^{n} \#(\theta_{i+1}\alpha_j) \\ &> \sum_{j=1}^{n} \#(\theta_{i+1}\alpha_j) \end{aligned}$$

which together with the fact $TV(S_{i+1}) = (TV(S_i) \setminus \{\alpha\}) \cup \{\alpha_1, \dots, \alpha_n\}$ implies $|S_i|_{TV} > |S_{i+1}|_{TV}$. The reasoning for rule Expand-R is similar.

If rule UNIFY is applied, we first consider the case where no variable is assigned during unification. Then $TV(S_i) = TV(S_{i+1})$ and $\theta_i = \theta_{i+1}$ hold and imply $|S_i|_{TV} = |S_{i+1}|_{TV}$. Furthermore, $|S_i| > |S_{i+1}|$ holds because a constraint gets removed. The other case is that some set of variables $\overline{\alpha}$ gets assigned. Then $TV(S_{i+1}) = TV(S_i) \setminus \overline{\alpha}$ and the fact that $\theta_i$ and $\theta_{i+1}$ only differ concerning the variables from the set $\overline{\alpha}$ implies $|S_i|_{TV} > |S_{i+1}|_{TV}$.

In all cases it holds either $|S_i|_{TV} > |S_{i+1}|_{TV}$ or $|S_i|_{TV} = |S_{i+1}|_{TV}$ and $|S_i| > |S_{i+1}|$. Thus $\forall i > 0.\ (|S_i|_{TV}, |S_i|) > (|S_{i+1}|_{TV}, |S_{i+1}|)$ holds.

*Termination of* $\Longrightarrow^!_{cyc}$ The application of the CYCLE-ELIM rule decreases the number of cycles of the constraint graph. The normal form is reached when this number is zero, that is the graph is a DAG.

*Termination of* $\Longrightarrow^!_{sol}$ The application of the ASSIGN-SUP and ASSIGN-INF rules decreases the number of unassigned variables in the constraint graph. The normal form is reached when type variables only occur in weakly connected components of the constraint graph that don't contain base types.

*Termination of* $\Longrightarrow^!_{unif}$ The application of the UNIFY-WCC rule decreases the number of vertices of the remaining constraint graph. The normal form is reached when the remaining graph is empty. □

## 6 Completeness

In this section we provide a completeness result for the algorithm described in §3. For the extension to type classes, described in §4, we don't have a similar result, without certain assumptions on the quasi-order of sorts. Since our goal was to build the coercion inference on top of Isabelle without changing the underlying type system, we will not embed such assumptions on the sort structure in our reasoning about the algorithm.

### 6.1 Some examples for incompleteness

So far, we have only made statements about termination and correctness of our algorithm. It is equally important that the algorithm does not fail for a term that can be coerced to a well-typed term. An algorithm with this property is called complete. As mentioned earlier, our algorithm is not complete for arbitrary posets of base types. This is shown by some simple examples. First, we consider the algorithm for the language without type classes as described in §3.

*Example 4.* Figure 17 shows a constraint graph and base type order where our algorithm may fail, although $\{\alpha \mapsto \mathbb{C},\ \beta \mapsto \mathbb{N}\}$ is a solving substitution. If during constraint resolution the type variable $\alpha$ is assigned first, it will receive value $\mathbb{R}$. Then, the assignment of $\beta$ will fail, since the infimum $\mathbb{R} \sqcap \mathbb{N}$ does not exist in the given poset. The fact that our algorithm does find the solution if $\beta$ is assigned before $\alpha$ is practically irrelevant because we cannot possibly exhaust all nondeterministic choices.

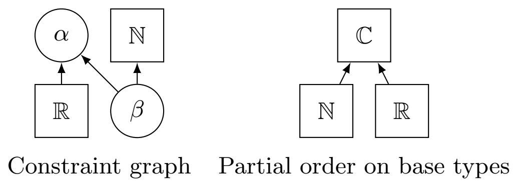


**Fig. 17.** Problematic example without type classes

Before we reason about a possible solution of the completeness problem we consider a second example now involving type classes.

*Example 5.* Some problems also occur in the solvable constraint graph in Figure 18. $\{\alpha \mapsto \mathbb{Q}\}$ is a solving substitution. The presented algorithm tries to assign $\alpha$ to $\bigsqcup^{\emptyset}_{Field}\{\mathbb{N}\}$. This supremum does not exist, since $\mathbb{R} \not<: \mathbb{Q}$ and $\mathbb{Q} \not<: \mathbb{R}$. Thus, the algorithm fails.

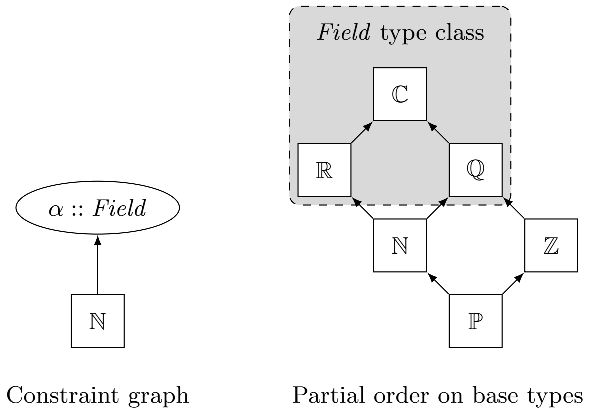


**Fig. 18.** Problematic example with type classes

In both cases, the problem is the non-existence of a supremum or infimum. The solution to this problem in the case without sorts is to require a certain lattice structure for the partial order on base types. Alternatively we could try and generalize our algorithm, but this is unappealing for complexity theoretic reasons.

## 6.2 Complexity and completeness

Tiuryn and Frey [18,4] showed that the general constraint satisfaction problem is PSPACE-complete. Tiuryn [18] also shows that satisfiability can be tested in polynomial time if the partial order on base types is a disjoint union of lattices.

Unfortunately, Tiuryn only gives a decision procedure that does not compute a solution. Nevertheless, most if not all approaches in the literature adopt the restriction to (disjoint unions of) lattices, but propose algorithms that are exponential in the worst case. This paper is no exception. Just like Simonet [17] we argue that in practice the exponential nature of our algorithm does not show up. Our implementation in Isabelle confirms this.

All phases of our algorithm have polynomial complexity except for constraint simplification: a cascade of applications of Expand-L or Expand-R may produce an exponential number of new type variables. Restricting to disjoint union of lattices does not improve the complexity but guarantees completeness of our algorithm because it guarantees the existence of the necessary infima and suprema for constraint resolution.

## 6.3 Proof of completeness in the case without type classes

In the following proof we assume that the base type poset is a disjoint union of lattices.

To formulate the completeness theorem, we need some further notation.

**Definition 10 (Equality modulo coercions).** *Two substitutions $\theta$ and $\theta'$ are* equal modulo coercions *wrt. the type variable set $X$, if for all $x \in X$ there exists a coercion $c$ such that either $\theta(x) <:_c \theta'(x)$ or $\theta'(x) <:_c \theta(x)$ holds. We write $\theta \approx_X \theta'$.*

**Definition 11 (Subsumed).** *The substitution $\theta'$ is* subsumed modulo coercions *wrt. to the type variable set $X$ by the substitution $\theta$, if there exists a substitution $\delta$ such that $\theta' \approx_X \delta \circ \theta$. We write $\theta \lesssim_X \theta'$.*

**Definition 12 (Invariant).** *The constraint set $S$ is* invariant *under the substitution $\theta$ if for all $\tau \lhd \sigma \in S$ it holds $\theta\tau = \tau$ and $\theta\sigma = \sigma$.*

*The constraint graph $G$ is* invariant *under the substitution $\theta$ if for all $\alpha \in TV(G)$ it holds $\theta\alpha = \alpha$.*

Let $TV(\tau)$ and $TV(t)$ be the sets of type variables that occur in $\tau$ and the type annotations of $t$. For a context $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, we denote by $TV(\Gamma)$ the set $\bigcup_{i=1}^{n} TV(\tau_i)$.

**Theorem 4 (Completeness).** *If $\theta'\Gamma \vdash \theta' t \rightsquigarrow u : \tau'$, then $AlgSol(\Gamma, t) \neq \emptyset$ and for all $\theta \in AlgSol(\Gamma, t)$ it holds that $\theta \lesssim_{TV(\Gamma) \cup TV(t)} \theta'$.*

Again, we reformulate the above theorem to have a closer look at the internals of our algorithm.

**Theorem 5 (Completeness).** *Suppose that $\theta'\Gamma \vdash \theta' t \rightsquigarrow u : \tau'$ and $\Gamma \vdash t : \tau \rhd S$. Then, there exist $S', G, G', \theta_{simp}, \theta_{cyc}, \theta_{sol}, \theta$ such that:*

$$\tau' = \theta'\tau, \tag{8}$$

$$S \text{ is weakly unifiable}, \tag{9}$$

$$(S, \emptyset) \Longrightarrow^!_{simp} (S', \theta_{simp}), \tag{10}$$

$$(G(S'), \theta_{simp}) \Longrightarrow^!_{cyc} (G, \theta_{cyc}), \tag{11}$$

$$(G, \theta_{cyc}) \Longrightarrow^!_{sol} (G', \theta_{sol}), \tag{12}$$

$$(G'[TV(G')], \theta_{sol}) \Longrightarrow^!_{unif} ((\emptyset, \emptyset), \theta). \tag{13}$$

*Further, for any such $S', G, G', \theta_{simp}, \theta_{cyc}, \theta_{sol}, \theta$, independent of the nondeterministic choices of the algorithm, it holds:*

$$\theta \lesssim_{TV(\Gamma)\cup TV(t)} \theta'. \tag{14}$$

*Proof.* From Lemma 9 and Lemma 10 we obtain (8), (9) and a substitution $\delta$ such that $\delta \circ \theta'$ solves $S$. By induction on the transitive closure of $\Longrightarrow_{simp}$, we obtain (10) and substitutions $\delta'$ and $\delta_{simp}$ such that $\delta \circ \theta' = \delta_{simp} \circ \theta_{simp}$ and $\delta' \circ \delta \circ \theta'$ solves $S'$, and therefore $G(S')$ using Lemma 11 and Lemma 12 for the induction step. By induction on the transitive closure of $\Longrightarrow_{cyc}$, we obtain (11) and a substitution $\delta_{cyc}$ such that $\delta' \circ \delta \circ \theta' = \delta_{cyc} \circ \theta_{cyc}$ and $\delta' \circ \delta \circ \theta'$ solves $G$ using Lemma 13 and Lemma 14 for the induction step. From both inductions, we also know that $\delta$ affects only variables that were introduced in the constraint generation and $\delta'$ affects only variables that were introduced in the constraint simplification. Thus, $\delta' \circ \delta \circ \theta'$ restricted to the free variables of the input $TV(\Gamma) \cup TV(t)$ equals the substitution $\theta'$. Thus $\theta' \approx_{TV(\Gamma)\cup TV(t)} \delta_{cyc} \circ \theta_{cyc}$. By induction on the transitive closure of $\Longrightarrow_{sol}$, we obtain (12) and a substitution $\delta_{sol}$ such that $\delta_{cyc} \circ \theta_{cyc} \approx_{TV(\Gamma)\cup TV(t)} \delta_{sol} \circ \theta_{sol}$ and $\delta_{sol} \circ \theta_{sol}$ solves $G'$, and therefore $G'[TV(G')]$ using Lemma 15 and Lemma 16 for the induction step. By induction on the transitive closure of $\Longrightarrow_{unif}$, we obtain (13) and a substitution $\delta_{unif}$ such that $\delta_{sol} \circ \theta_{sol} \approx_{TV(\Gamma)\cup TV(t)} \delta_{unif} \circ \theta$ using Lemma 17 and Lemma 18 for the induction step. Finally, we have $\theta' \approx_{TV(\Gamma)\cup TV(t)} \delta_{unif} \circ \theta$, and therefore (14). $\square$

**Lemma 9.** *Suppose that $\theta'\Gamma \vdash \theta' t \rightsquigarrow u : \tau'$ and $\Gamma \vdash t : \tau \rhd S$. Then there exists a substitution $\delta$ such that $\tau' = (\delta \circ \theta')\tau$ and $\delta \circ \theta'$ solves $S$. Moreover, $\delta$ is a substitution that affects only fresh variables that were generated in the constraint generation.*

*Proof.* By induction on the constraint generation for an arbitrary $\theta'$.

SubCT-Var $(x : \tau \in \Gamma)$ Using $\theta'\Gamma \vdash \theta' x \rightsquigarrow u : \tau'$, we derive that $\tau' = \theta'\tau$. Moreover, every substitution solves the generated empty constraint set. Thus, the choice $\delta = \emptyset$ proves this base case.

SubCT-Const $(\Sigma(c) = \sigma)$ Using $\theta'\Gamma \vdash \theta' c_{[\overline{a} \mapsto \overline{\tau}]} \rightsquigarrow u : \tau'$, we derive that $\tau' = \theta'\,(\sigma[\overline{a} \mapsto \overline{\tau}])$. Again, $\theta'$ solves the generated empty constraint set. Thus, the choice $\delta = \emptyset$ also proves this base case.

SubCT-Abs $(\Gamma, x : \tau_1 \vdash t : \sigma \rhd S)$ Since the judgement $\theta'\Gamma \vdash \theta'(\lambda x : \tau_1.\ t) \rightsquigarrow u : \tau'$ must have been derived via the Coerce-Abs rule, there exist $\tau_2$ and $u'$ such that $\theta'(\Gamma, x : \tau_1) \vdash \theta' t \rightsquigarrow u' : \tau_2$ and $\tau' = \theta'\tau_1 \to \tau_2$. From the induction hypothesis, we obtain a $\delta$ such that $\delta \circ \theta'$ solves the constraint set $S$ and $\tau_2 = (\delta \circ \theta')\sigma$. The rule SubCT-Abs does not generate any new constraints. Also, $\delta$ affects only variables that are introduced by the constraint generation. Thus, $(\delta \circ \theta')\tau_1 = \theta'\tau_1$. It follows that $\tau' = (\delta \circ \theta')(\tau_1 \to \sigma)$ and $\delta \circ \theta'$ solves the same constraint set $S$ that is generated from the term $\lambda x : \tau_1.\ t$ in the context $\Gamma$.

SubCT-App $(\Gamma \vdash f : \tau \rhd S_f, \Gamma \vdash t : \sigma \rhd S_t)$ Since the judgement $\theta'\Gamma \vdash \theta'(f\ t) \rightsquigarrow u : \tau'$ must have been derived via the Coerce-App rule, there exist $\tau_f$, $u_f$, $\tau_t$, $u_t$, and $c$ such that $\theta'\Gamma \vdash \theta' f \rightsquigarrow u_f : \tau_f \to \tau'$, $\theta'\Gamma \vdash \theta' t \rightsquigarrow u_t : \tau_t$ and $\tau_t <:_c \tau_f$. From the induction hypothesis for $f$, we obtain a $\delta_f$ such that $\delta_f \circ \theta'$ solves $S_f$, $\tau_f \to \tau' = (\delta_f \circ \theta')\tau$. Since $\delta_f$ affects only variables that are introduced in the derivation of $S_f$, $\theta'\Gamma$ and $\theta' t$ are invariant under the application of $\delta_f$. Thus, $(\delta_f \circ \theta')\Gamma \vdash (\delta_f \circ \theta')t \rightsquigarrow u_t : \tau_t$ holds. From the induction hypothesis for $t$, we obtain a $\delta_t$ such that $\delta_t \circ \delta_f \circ \theta'$ solves $S_t$ and $\tau_t = (\delta_t \circ \delta_f \circ \theta')\sigma$. Finally, we need to solve the newly generated constraints $\{\tau \doteq \alpha \to \beta,\ \sigma <: \alpha\}$ providing an adequate substitution of the fresh variables $\alpha$ and $\beta$. Let $\delta = \{\alpha \mapsto \tau_f, \beta \mapsto \tau'\}$. Then, together with the fact that $\delta_t$ affects only variables that are introduced in the derivation of $S_t$, we know that $(\delta \circ \delta_t \circ \delta_f) \circ \theta'$ solves $S_f \cup S_t \cup \{\tau \doteq \alpha \to \beta,\ \sigma <: \alpha\}$. This is the case, because $\delta_f \circ \theta'$ already solves $S_f$, $\delta_t \circ \delta_f \circ \theta'$ already solves $S_t$, $(\delta \circ \delta_t \circ \delta_f \circ \theta')\tau = \tau_f \to \tau' = (\delta \circ \delta_t \circ \delta_f \circ \theta')(\alpha \to \beta)$ and $(\delta \circ \delta_t \circ \delta_f \circ \theta')\sigma = \tau_t <:_c \tau_f = (\delta \circ \delta_t \circ \delta_f \circ \theta')\alpha$. Moreover, $((\delta \circ \delta_t \circ \delta_f) \circ \theta')\beta = \tau'$ such that $\delta \circ \delta_t \circ \delta_f$ fulfils the needed properties. □

**Lemma 10 (Weak unifiability).** *If $S$ has a solution, then $S$ is weakly unifiable.*

*Proof.* Assuming $\tau <:_c \sigma$, a straightforward induction on the coercion generation provides the result $\lceil \tau \rceil = \lceil \sigma \rceil$.

Suppose $\theta$ solves $S$. Hence, for all $\tau <: \sigma \in S$ there exists a coercion $c$ such that it holds $\theta\tau <:_c \theta\sigma$, and therefore $\lceil \theta\tau \rceil = \lceil \theta\sigma \rceil$. Since $\theta$ also solves all equality constraints in $S$, it is a weak unifier of $S$. □

**Lemma 11 (Progress of $\Longrightarrow_{simp}$).** *Suppose that $\theta''$ solves $S$. Then either $S$ is atomic or for all $\theta$ there exist $S', \theta'$ such that $(S, \theta) \Longrightarrow_{simp} (S', \theta')$.*

*Proof.* Assume $S$ is not atomic. Since our subtyping is structural and $S$ is solvable, constraints of the forms, $C\ \tau_1\ \ldots\ \tau_n <: T$, $T <: C\ \tau_1\ \ldots\ \tau_n$, and $C\ \tau_1\ \ldots\ \tau_n <: D\ \tau_1\ \ldots\ \tau_m$ where $n > 0$, $m > 0$, and $C \neq D$ do not occur in $S$. Thus, there are five kinds of constraints in $S$, each kind corresponding to a simplification rule. To prove the possibility of making a reduction step with $\Longrightarrow_{simp}$, we have to show that the side conditions of the rules hold. Those are the existence of the most general unifier of $\tau$ and $\sigma$ in the rule Unify and the compatibility of the base types $T$ and $U$ with the subtype relation in the rule Eliminate. Both hold since $S$ is solvable. □

**Lemma 12 (Preservation of solvability by $\Longrightarrow_{simp}$).** *Suppose that $(S, \theta) \Longrightarrow_{simp} (S', \theta')$, $\theta''$ solves $S$, there exists $\delta$ such that $\theta'' = \delta \circ \theta$, and $S$ is invariant under $\theta$.*

*Then there exists a $\delta', \delta''$ such that $\theta'' = \delta' \circ \theta'$, $\delta'' \circ \theta''$ solves $S'$, and $S'$ is invariant under $\theta'$. Moreover, $\delta''$ is a substitution that affects only fresh variables that were generated in that particular step of constraint simplification.*

*Proof.* By case distinction on the constraint generation.

Decompose Since $\theta'' = \delta \circ \theta$ solves $\{C\ \tau_1\ \dots\ \tau_n <: C\ \sigma_1\ \dots\ \sigma_n\} \uplus S$, by definition of *var* it also solves $\{var^i_C(\tau_i, \sigma_i) \mid i = 1 \dots n\} \cup S$. Moreover, $\{var^i_C(\tau_i, \sigma_i) \mid i = 1 \dots n\}$ contains exactly the same type variables as $\{C\ \tau_1\ \dots\ \tau_n <: C\ \sigma_1\ \dots\ \sigma_n\}$ such that $\{var^i_C(\tau_i, \sigma_i) \mid i = 1 \dots n\} \cup S$ is invariant under $\theta$. Finally, with $\theta' = \theta$ the case follows for $\delta'' = \emptyset$, $\delta' = \delta$.

Unify Since $\theta'' = \delta \circ \theta$ solves $\{\tau \doteq \sigma\} \uplus S$ and $\{\tau \doteq \sigma\} \uplus S$ is invariant under $\theta$, it holds $\delta\tau = \theta''\tau = \theta''\sigma = \delta\sigma$. Thus, $\delta$ is a unifier for $\{\tau, \sigma\}$ and we obtain a $\delta_1$ where $\delta = \delta_1 \circ mgu\{\tau, \sigma\} = \delta' \circ \theta'$. It follows that $\theta'' = \delta \circ \theta = \delta_1 \circ (\theta' \circ \theta)$.

We can assume the idempotence of the most general unifier such that $\theta' S$ is invariant under $\theta'$. Further, the range of $\theta'$ is a subset of $TV(\tau) \cup TV(\sigma)$, such that the type variables of $\theta' S$ are a subset of the type variables of $\{\tau \doteq \sigma\} \uplus S$. Since $\{\tau \doteq \sigma\} \uplus S$ is invariant under $\theta$, $\theta' S$ must also be invariant under $\theta$. Thus, $\theta' S$ is invariant under $\theta' \circ \theta$.

Moreover, with $\theta(\theta' S) = \theta' S$, the idempotence of $\theta'$ and $\theta S = S$, it follows:

$$\begin{aligned}
\theta''(\theta' S) &= (\delta_1 \circ \theta' \circ \theta)(\theta' S)\\
&= (\delta_1 \circ \theta')(\theta' S)\\
&= (\delta_1 \circ \theta') S\\
&= (\delta_1 \circ \theta' \circ \theta) S\\
&= \theta'' S
\end{aligned}$$

Hence, as $\theta''$ solves $S$, it also solves $\theta' S$. The case follows for $\delta'' = \emptyset$, $\delta' = \delta_1$.

Expand-L Since our subtyping is structural, $\theta'' = \delta \circ \theta$ solves $\{\alpha <: C\ \tau_1\ \dots\ \tau_n\}$ and $\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \uplus S$ is invariant under $\theta$, there exist $\sigma_1 \dots \sigma_n$ where $\theta''\alpha = \delta\alpha = C\ \sigma_1\ \dots\ \sigma_n$. Let $\delta_1$ be the substitution $\{\alpha_i \mapsto \sigma_i \mid i = 1 \dots n\}$. As the $\alpha_i$ are freshly introduced by the algorithm, we can rewrite $\delta$ as an instance of $\theta' = \{\alpha \mapsto C\ \alpha_1\ \dots\ \alpha_n$. We obtain a $\delta_2$ such that $\delta = \delta_2 \circ \delta_1 \circ \theta'$, and therefore $\theta'' = \delta \circ \theta = \delta_2 \circ \delta_1 \circ (\theta' \circ \theta)$. Further, it is easy to see that $\delta_1 \circ \theta''$ solves $\theta'(\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \cup S)$ using once again the facts that $\alpha_i$ are fresh variables and $\theta''$ solves $\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \uplus S$.

Finally, we have to show that $\theta'(\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \cup S)$ is invariant under $\theta' \circ \theta$. Since the variables in the range of $\theta'$ are fresh, and therefore unaffected by $\theta$, it is enough to show that $\theta' \circ \theta(\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \cup S)$ is invariant under $\theta'$. From the fact that $\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \cup S$ is invariant under $\theta$, it follows that $\theta' \circ \theta(\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \cup S) = \theta'(\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \cup S)$. Clearly, $\theta'$ is idempotent such that $\theta'(\{\alpha <: C\ \tau_1\ \dots\ \tau_n\} \cup S)$ is invariant under $\theta'$. The case follows for $\delta' = \delta_2 \circ \delta_1$ and $\delta'' = \delta_1$.

Expand-R Analogously to the Expand-L case.

Eliminate Since $\theta'' = \delta \circ \theta$ solves $\{U <: T\} \uplus S$, it of course also solves just $S$. Further, with $\theta' = \theta$ the case follows for $\delta'' = \emptyset$, $\delta' = \delta$. $\square$

**Lemma 13 (Progress of $\Longrightarrow_{cyc}$).** *Suppose that $\theta''$ solves $G$. Then either $G$ is a DAG or for all $\theta$ there exist $G', \theta'$ such that $(G, \theta) \Longrightarrow_{cyc} (G', \theta')$.*

*Proof.* Assume $G$ is not a DAG. Thus, there is a cycle $K = \{\tau_1 \dots \tau_n\}$ in $G$. Since $\theta''$ solves $G$, it must hold $\theta''\tau_1 \prec: \dots \prec: \theta''\tau_n \prec: \theta''\tau_1$, and therefore $\theta''\tau_1 = \dots = \theta''\tau_n$. From this, we know that the computation of $mgu(K)$ in the Cycle-Elim rule will not fail and we obtain the transformed graph $G'$ and the new substitution $\theta' = mgu(K) \circ \theta$. $\square$

**Lemma 14 (Preservation of solvability by $\Longrightarrow_{cyc}$).** *Suppose that $(G, \theta) \Longrightarrow_{cyc} (G', \theta')$, $\theta''$ solves $G$, there exists $\delta$ such that $\theta'' = \delta \circ \theta$, and $G$ is invariant under $\theta$.*

*Then there exists a $\delta'$ such that $\theta'' = \delta' \circ \theta'$, $\theta''$ solves $G'$, and $G'$ is invariant under $\theta'$.*

*Proof.* By a degenerated case distinction on the cycle elimination.

Cycle-Elim Let $K = \{\tau_1 \dots \tau_n\}$ be the eliminated cycle in $G$. As $\theta''$ solves $G$, it must hold $\theta''\tau_1 = \dots = \theta''\tau_n$. Then, it follows that $\theta''$ is also a solution for $G'$ by construction of $E'$, $P$ and $S$.

Since $G$ is invariant under $\theta$, $\theta'' = \delta \circ \theta$ and it holds $\theta''\tau_1 = \dots = \theta''\tau_n$, we have $\delta\tau_1 = \dots = \delta\tau_n$. Hence, $\delta$ is a unifier for $K$. For the most general unifier $\theta_K = mgu(K)$ we obtain a $\delta'$ where $\delta = \delta' \circ \theta_K$. Thus, $\theta'' = \delta \circ \theta = \delta' \circ \theta_K \circ \theta = \delta' \circ \theta'$.

Finally, we have to show that $G'$ is invariant under $\theta' = \theta_K \circ \theta$. $G'$ is invariant under $\theta$, since $TV(G') \subseteq TV(G)$ and $G$ is invariant under $\theta$. Moreover, $G'$ is invariant under $\theta_K$, since $\theta_K\tau_K = \tau_K$ by idempotence of the most general unifier and for all $\alpha \in TV(V \setminus K)$ it holds $\theta_K\alpha = \alpha$. The latter is the case, because there would be a more general unifier than $\theta_K$, if it would affect variables that are not in $K$. Ultimately, it follows that $G'$ is invariant under $\theta'$. $\square$

**Lemma 15 (Progress of $\Longrightarrow_{sol}$).** *Suppose that $G$ is a DAG and $\theta''$ solves $G$. Then either for all type variables $\alpha \in TV(G)$ it holds $P_\alpha \cup S_\alpha = \emptyset$ or for all $\theta$ there exist $G', \theta'$ such that $(G, \theta) \Longrightarrow_{sol} (G', \theta')$.*

*Proof.* Assume that there is an $\alpha$ such that $P_\alpha \cup S_\alpha \neq \emptyset$. Without loss of generality, we may assume that $P_\alpha \neq \emptyset$. Since $\theta''$ is a solution for $G$, for all $T \in S_\alpha$ and for all $S \in P_\alpha$ it holds $\theta''\alpha \prec: T$ and $S \prec: \theta''\alpha$. It follows that $\theta''\alpha \in \bigcap_{T \in P_\alpha} \overline{T}$. Then, all types in $\bigcap_{T \in P_\alpha} \overline{T}$ must belong to a single weakly connected component of the base type poses, because otherwise this would contradict the existence of common subtypes in $P_\alpha$. Together with our assumption about the base type poset being a disjoint union of lattices, $\bigcap_{T \in P_\alpha} \overline{T}$ must have a smallest element. Hence, $\bigsqcup P_\alpha$ exists. Further, for all $T \in S_\alpha$ it holds $\bigsqcup P_\alpha \prec: \theta''\alpha \prec: T$. Thus, the rule Assign-Sup and not Fail-Sup is applicable and provides us the necessary $G'$ and $\theta'$. □

**Lemma 16 (Preservation of solvability by $\Longrightarrow_{sol}$).** *Suppose that $(G, \theta) \Longrightarrow_{sol} (G', \theta')$, $\delta \circ \theta$ solves $S$ and $G$ is invariant under $\theta$.*

*Then there exists a $\delta'$ such that $\delta \circ \theta \approx_X \delta' \circ \theta'$, $\delta' \circ \theta'$ solves $G'$ and $G'$ is invariant under $\theta'$.*

*Proof.* By case distinction on the constraint resolution.

Assign-Sup For the assigned variable $\alpha \in TV(G)$, we know that $P_\alpha \neq \emptyset$, $\exists \bigsqcup P_\alpha$ and for all $T \in S_\alpha$ it holds $\bigsqcup P_\alpha <: T$.

Clearly, $\theta' = \{\alpha \mapsto \bigsqcup P_\alpha\}$ is idempotent, and therefore $\theta' G$ invariant under $\theta'$. Further, $\theta' G$ is invariant under $\theta$ because $G$ is invariant under $\theta$ and $TV(\theta' G) \subseteq TV(G)$. Thus, $\theta' G$ is invariant under $\theta' \circ \theta$.

Since $G$ is invariant under $\theta$ and $\delta \circ \theta$ solves $G$, $\delta$ also solves $G$.

Let $G = (V, E)$. We define the substitution $\delta'$ as following:

$$\delta'\beta = \begin{cases} \delta\beta \sqcap \bigsqcup P_\alpha & \text{if } (\beta, \alpha) \in E^+ \\ \delta\beta & \text{otherwise} \end{cases}$$

This definition makes sense, because the types $\delta\beta$ and $\bigsqcup P_\alpha$ belong to the same weakly connected component of the base type poset (which is, as we required, a disjoint union of lattices), and therefore their infimum exists.

From the definition of the supremum and the fact that $(\delta \circ \theta)\alpha$ must be an upper bound for $P_\alpha$ to solve the constraint graph, we have $(\delta' \circ \theta' \circ \theta)\alpha = \bigsqcup P_\alpha \prec: (\delta \circ \theta)\alpha$. Together with the definition of $\delta'$, for all $X$ it holds $\delta \circ \theta \approx_X \delta' \circ \theta' \circ \theta$.

Furthermore, we claim that $\delta' \circ \theta' \circ \theta$ solves $G'$. To prove this, it is enough to show that $\delta'$ solves $G'$. $\delta'$ operates on constraints between vertices of $G'$, that do not contain a type variable predecessor of $\alpha$ in exactly the same way as $\delta$, such that for those the solvability is preserved.

Assume that $\beta$ is a type variable predecessor of $\alpha$, i.e. $(\beta, \alpha) \in E^+$. Now we show that all the constraints involving $\beta$ are fulfilled. There are four different kinds of constraints that we have to consider: constraints between the new assignment of $\alpha$ and $\beta$, constraints between predecessors of $\alpha$, i.e. $(\beta', \beta) \in E^+$, constraints between $\beta$ and a base type predecessor $T$ of $\beta$, i.e. $(T, \beta) \in E^+$, and constraints between $\beta$ and successors $\tau$ of $\beta$, i.e. $(\beta, \tau) \in E^+$, where $\tau$ is either a base type $U$ or a type variable $\gamma$ that is not a predecessor of $\alpha$. Constraints of the form $(\beta, \alpha) \in E^+$ are solved by $\delta'$, since $\alpha$ is assigned the type $\bigsqcup P_\alpha$, and $\delta'\beta = \delta\beta \sqcap \bigsqcup P_\alpha \prec: \bigsqcup P_\alpha$. To see why constraints of the form $(\beta', \beta) \in E^+$ are solved by $\delta'$, note that $\delta\beta' \prec: \delta\beta$, otherwise $\delta$ would not have been a solution of $G$. Therefore, we have that $\delta\beta' \sqcap \bigsqcup P_\alpha \prec: \delta\beta \sqcap \bigsqcup P_\alpha$ by monotonicity, which, by definition of $\delta'$, is equivalent to $\delta'\beta' \prec: \delta'\beta$. Constraints of the form $(T, \beta) \in E^+$ still hold, because $T \in P_\beta \subseteq P_\alpha$, and therefore $T \prec: \bigsqcup P_\alpha$. Since $\delta$ is a solution of the original constraint set, we also have that $T \prec: \delta\beta$, so it follows that $T \prec: \delta\beta \sqcap \bigsqcup P_\alpha$. Finally, constraints of the form $(\beta, \tau) \in E^+$ are still solved by $\delta'$, because $\delta'\beta \prec: \delta\beta$, and either $\delta\beta \prec: U$ if $\tau = U$, or $\delta'\gamma = \delta\gamma$ and $\delta\beta \prec: \delta\gamma$ if $\tau = \gamma$, so $\delta'\beta \prec: \delta'\tau$ trivially holds.

Thus, our defined $\delta'$ proves this case.

Assign-Inf Analogously to the Assign-Sup case. □

**Lemma 17 (Progress of $\Longrightarrow_{unif}$).** *Suppose that all vertices of $G$ are type variables and $\theta''$ solves $G$. Then either $G = (\emptyset, \emptyset)$ or for all $\theta$ there exist $G', \theta'$ such that $(G, \theta) \Longrightarrow_{sol} (G', \theta')$.*

*Proof.* Assume $G \neq (\emptyset, \emptyset)$. Then, there exists a non-empty weakly connected component $W$ of $G$. $W$ is unifiable, because it consists only of type variables. Thus, the application of the rule Unify-WCC does not fail. □

**Lemma 18 (Preservation of solvability by $\Longrightarrow_{unif}$).** *Suppose that $(G, \theta) \Longrightarrow_{unif} (G', \theta')$, $\delta \circ \theta$ solves $G$ and $G$ is invariant under $\theta$.*

*Then there exists a $\delta'$ such that $\delta \circ \theta \approx_X \delta' \circ \theta'$, $\delta' \circ \theta'$ solves $G'$ and $G'$ is invariant under $\theta'$.*

*Proof.* By a degenerated case distinction on the unification of weakly connected components.

Unify-WCC $G' = G[V \setminus W]$ is invariant under $\theta$, since $G$ is invariant under $\theta$. Moreover, $G'$ is invariant under $mgu(W)$, since $mgu(W)$ only affects variables in $W$. Thus, $G'$ is invariant under $\theta' = mgu(W) \circ \theta$.

Let $\{\alpha_W\} = mgu(W)W$. Then, $\alpha_W \in W$ must hold. Since our subtyping is structural and $\delta$ solves $W$ and the base type poset is a disjoint union of lattices, either all types in $\delta W$ are equal to a single type variable from $W$ or they are all base types. Define

$$\delta' = \left\{ \alpha_W \mapsto \begin{cases} \alpha'_W & \delta W = \{\alpha'_W\} \\ \bigsqcup(\delta W) & \text{otherwise} \end{cases} \right\}$$

Now, we can show that for all $X$ it holds $\delta \circ \theta \approx_X \delta \circ \delta' \circ \theta'$. Assume that $\alpha \in X$. If $\alpha \notin W$ holds, then $(\delta \circ \theta)\alpha = \delta\alpha = (\delta \circ \delta' \circ \theta')\alpha$ holds using the above invariance statements. Otherwise, it holds $(\delta \circ \theta)\alpha = \delta\alpha$ and $(\delta \circ \delta' \circ \theta')\alpha = \delta(\delta'\alpha_W)$. Now, if $\delta W = \{\alpha'_W\}$ holds, then $\delta(\delta'\alpha_W) = \alpha'_W = \delta\alpha$ holds for all $\alpha \in W$ by definition of $\delta'$. Otherwise, $\delta(\delta'\alpha_W) = \bigsqcup(\delta W) \prec: \delta\alpha$ holds for all $\alpha \in W$ by the definitions of $\delta'$ and the supremum. Thus, $\delta \circ \theta \approx_X (\delta \circ \delta') \circ \theta'$.

Further, using the above invariance statements once again, we have $(\delta \circ \delta' \circ \theta')G' = (\delta \circ \delta' \circ mgu(W) \circ \theta)G[V \setminus W] = (\delta \circ \theta)G[V \setminus W]$. Together with the fact that $\delta \circ \theta$ solves $G$, and therefore $G[V \setminus W]$, this implies that $(\delta \circ \delta') \circ \theta'$ solves $G'$. □

It is instructive to consider a case where our algorithm is not able to reconstruct a particular substitution but only a subsumed one.

*Example 6.* Let $\Sigma = \{id : \alpha \to \alpha, n : \mathbb{N}, sin : \mathbb{R} \to \mathbb{R}\}$ be a signature and let $\mathcal{C} = \{int : \mathbb{N} \to \mathbb{Z}, real : \mathbb{Z} \to \mathbb{R}\}$ be a set of coercions. Now consider the term $sin\ (id_{[\alpha \mapsto \alpha_1]}\ n)$ in the empty context. The constraint resolution phase will be given the atomic constraints $\{\mathbb{N} <: \alpha_1, \alpha_1 <: \mathbb{R}\}$ and will assign $\alpha_1$ the tightest bound either with respect to its predecessors or its successors: $AlgSol(\emptyset, sin\ (id_{[\alpha \mapsto \alpha_1]}\ n)) = \{\{\alpha_1 \mapsto \mathbb{N}\}, \{\alpha_1 \mapsto \mathbb{R}\}\}$.

The substitution $\{\alpha_1 \mapsto \mathbb{Z}\}$ is also solution of the typing problem, i.e. $\{\alpha_1 \mapsto \mathbb{Z}\}\emptyset \vdash \{\alpha_1 \mapsto \mathbb{Z}\}(sin\ (id_{[\alpha \mapsto \alpha_1]}\ n)) \rightsquigarrow sin\ (real\ (id_{[\alpha \mapsto \mathbb{Z}]}\ (int\ n))) : \mathbb{R}$. It is itself not a possible output of the algorithm, but it is subsumed modulo coercions by both of the substitutions that the algorithm can return.

The completeness theorem tells us that the algorithm never fails if there is a solution. The example shows us that the algorithm may fail to produce some particular solution. The completeness theorem also tells us that any solution is an instance of the computed solution, but only up to coercions. In practice this means that the user may have to provide some coercions (or type annotations) explicitly to obtain what she wants. This is not the fault of the algorithm but is unavoidable if the underlying type system does not provide native subtype constraints.

Compared with the work by Saïbi we have a completeness result. On the other hand he goes beyond coercions between atomic types, something we have implemented but not yet released. Luo also proves a completeness result, but his point of reference is a modified version of the Hindley-Milner system where coercions are inserted on the fly, which is weaker than our inference system. In most other papers the type system comes with subtype constraints built in (not an option for us) and unrestricted completeness results can be obtained.

## 6.4 Incompleteness of the case with type classes

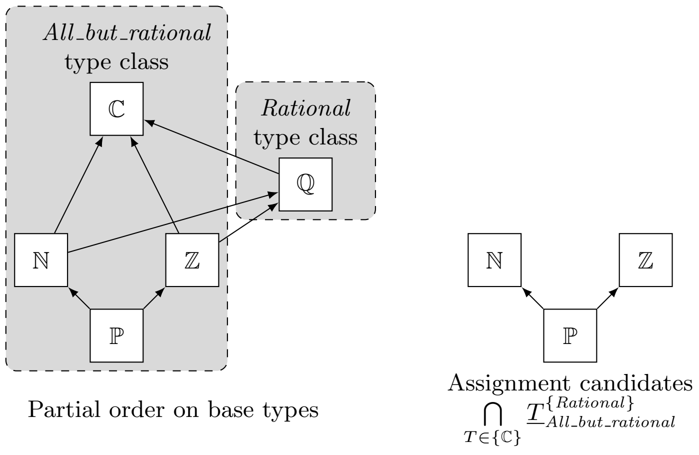
Example 5 demonstrates why a stronger restriction is necessary for the algorithm extended with type classes. In that example, the given poset of base types is a lattice but the supremum does not exist if we consider only base types that belong to the sort *Field*. The first idea to solve this problem is to require any restriction of our base type poset $(\mathcal{P}, \prec:)$ with any sort $\mathcal{S}$ to $(\mathcal{P}_\mathcal{S}, \prec:)$ where $\mathcal{P}_\mathcal{S} = \{\tau \in \mathcal{P} \mid \vdash_{sort} \tau : \mathcal{S}\}$ to be a disjoint union of lattices. This restriction extends Tiuryn's restriction, since every type belongs to the sort $\top$ so that $\mathcal{P} = \mathcal{P}_\top$ holds.

Unfortunately, this restriction is not strong enough. Example 7 demonstrates a problem that occurs even though the poset of base types restricted to any sort is a lattice. The reason for this is the fact that in the algorithm the poset of assignment candidates[2] is not only constrained with a single sort but also to the types that must have super-/subtypes belonging to some type classes. The structure of such a constrained poset could be almost random. At least, we can

[2] E.g. when we compute $\bigsqcap_\mathcal{S}^\mathcal{X} X$ the assignment candidates are $\bigcap_{T \in X} \underline{T}_\mathcal{S}^\mathcal{X}$

not expect the needed supremum/infimum to exist. In some sense, type class restrictions cut out arbitrary[3] type sets from the assignment candidates.

*Example 7.* Assume, the constraint resolution algorithm is applied to the following constraint set: $\{\alpha :: \mathit{All\_but\_rational} <: \mathbb{C},\ \alpha :: \mathit{All\_but\_rational} <: \beta :: \mathit{Rational}\}$



*All_but_rational* type class
*Rational* type class
$\mathbb{C}$
$\mathbb{Q}$
$\mathbb{N}$
$\mathbb{Z}$
$\mathbb{P}$
Partial order on base types
Assignment candidates $\bigcap_{T \in \{\mathbb{C}\}} \underline{T}^{\{Rational\}}_{All\_but\_rational}$

**Fig. 19.** Assignment candidates resulting from computation of $\bigsqcap^{\{Rational\}}_{All\_but\_rational}\{\mathbb{C}\}$

Figure 19 shows that the assignment candidates do not have a maximal element. So, the infimum does not exist and the algorithm fails.

Even if it will not lead us to a completeness result, let us consider a restriction of the base type poset, that guarantees the existence of a supremum/infimum for any set of assignment candidates, that can occur in a solvable constraint graph.

**Definition 13 (Sort-respecting subtype relation).** *We call a poset* $(\mathcal{P}, \prec:)$ sort-respecting *if for all* $A, B \in \mathcal{P}$ *it holds:*

- *if* $\vdash_{sort} A : \mathcal{S}$ *and* $\vdash_{sort} B : \mathcal{S}$*, then* $A \sqcap^{\emptyset}_{\mathcal{S}} B$ *exists, and for all* $U$ *such that* $\vdash_{sort} U : \mathcal{T}$*,* $\mathcal{T} \neq \mathcal{S}$ *and* $U \prec: A$*,* $U \prec: B$ *we have that* $U \prec: A \sqcap^{\emptyset}_{\mathcal{S}} B$*, and dually*
- *if* $\vdash_{sort} A : \mathcal{S}$ *and* $\vdash_{sort} B : \mathcal{S}$*, then* $A \sqcup^{\emptyset}_{\mathcal{S}} B$ *exists, and for all* $U$ *such that* $\vdash_{sort} U : \mathcal{T}$*,* $\mathcal{T} \neq \mathcal{S}$ *and* $A \prec: U$*,* $B \prec: U$ *we have that* $A \sqcup^{\emptyset}_{\mathcal{S}} B \prec: U$*.*

**Lemma 19 (Monotonicity I).** *Let A,B,X be base types from a sort-respecting poset with* $\vdash_{sort} A : \mathcal{S}$ *and* $\vdash_{sort} B : \mathcal{S}$*. Then it holds:*

- $A \prec: X \wedge B \prec: X \Rightarrow A \sqcup^{\emptyset}_{\mathcal{S}} B \prec: X$

[3] Actually, it is not completely arbitrary. If a type $\tau$ has no supertype in the class $\mathcal{S}$, then no supertype of $\tau$ has a supertype in the class $\mathcal{S}$. Nevertheless, the supremum/infimum could be cut out.

– $X \prec: A \wedge X \prec: B \Rightarrow X \prec: A \sqcap^{\emptyset}_{\mathcal{S}} B$

*Proof.* If $\vdash_{sort} X : \mathcal{S}$ both implications hold because of the minimality/maximality of the supremum/infimum within a sort. Otherwise the definition of sort-respecting gives us exactly the required. □

**Lemma 20 (Monotonicity II).** *Let A,B,X,Y be base types from a sort-respecting poset with $\vdash_{sort} A : \mathcal{S}$, $\vdash_{sort} B : \mathcal{S}$, $\vdash_{sort} X : \mathcal{T}$ and $\vdash_{sort} Y : \mathcal{T}$. Then it holds:*

$$A \prec: X \wedge B \prec: Y \Rightarrow A \sqcup^{\emptyset}_{\mathcal{S}} B \prec: X \sqcup^{\emptyset}_{\mathcal{T}} Y$$

*Proof.* Obviously, $A \prec: X \sqcup^{\emptyset}_{\mathcal{T}} Y$ and $B \prec: X \sqcup^{\emptyset}_{\mathcal{T}} Y$ hold. The claim follows immediately by Lemma 19. □

**Lemma 21 (Unambigousness).** *In a sort-respecting poset $(\mathcal{P}, \prec:)$, for all sets of base types $X \subseteq \mathcal{P}$, sets of type variables $\mathcal{X}$, and sorts $\mathcal{S}$ it holds:*

– *if* $\bigcap\limits_{T \in X} \underline{T}^{\mathcal{X}}_{\mathcal{S}} \neq \emptyset$ *then* $\bigcap\limits_{T \in X} \underline{T}^{\mathcal{X}}_{\mathcal{S}}$ *has a greatest element, and dually*
– *if* $\bigcap\limits_{T \in X} \overline{T}^{\mathcal{X}}_{\mathcal{S}} \neq \emptyset$ *then* $\bigcap\limits_{T \in X} \overline{T}^{\mathcal{X}}_{\mathcal{S}}$ *has a smallest element*

*Proof.* We show only one the first of the two symmetric claims. Assume $\bigcap\limits_{T \in X} \underline{T}^{\mathcal{X}}_{\mathcal{S}} \neq \emptyset$. If $\bigcap\limits_{T \in X} \underline{T}^{\mathcal{X}}_{\mathcal{S}}$ contains only one element, then we are done. Otherwise, let $A, B \in \bigcap\limits_{T \in X} \underline{T}^{\mathcal{X}}_{\mathcal{S}}$. By Definition 9, $\vdash_{sort} A : \mathcal{S}$, $\vdash_{sort} B : \mathcal{S}$ holds. In a sort-respecting poset the supremum $A \sqcup^{\emptyset}_{\mathcal{S}} B$ exists. Further, $\vdash_{sort} A \sqcup^{\emptyset}_{\mathcal{S}} B : \mathcal{S}$ and for all $T \in X$ it holds by monotonicity: $A \sqcup^{\emptyset}_{\mathcal{S}} B \prec: T$. Moreover, for all $\beta :: \mathcal{T} \in \mathcal{X}$ we have base types $U, V$ such that $A \prec: U, B \prec: V, \vdash_{sort} U : \mathcal{T}, \vdash_{sort} V : \mathcal{T}$. By Lemma 20 it holds $A \sqcup^{\emptyset}_{\mathcal{S}} B \prec: U \sqcup^{\emptyset}_{\mathcal{S}} V$. Thus, for all $\beta :: \mathcal{T} \in \mathcal{X}$ there exists a supertype of $A \sqcup^{\emptyset}_{\mathcal{S}} B$ of sort $\mathcal{T}$, namely $U \sqcup^{\emptyset}_{\mathcal{T}} V$. Hence $A \sqcup^{\emptyset}_{\mathcal{S}} B \in \bigcap\limits_{T \in X} \underline{T}^{\mathcal{X}}_{\mathcal{S}}$. But if for any two types from the intersection their supremum is contained in the intersection, then the intersection has a greatest element. □

The last lemma assures that in a sort-respecting poset of base types for every type variable of a solvable constraint set the supremum/infimum of its successors/predecessors is well-defined. Unfortunately, the local existence of a supremum/infimum, that was enough in the case without sorts, does not establish completeness of the alorithm with sorts. The following example demonstrates this.

*Example 8.* If the poset is a disjoint union of linear orders, it is also sort-respecting because cutting out a set of types from a linear order results again in a linear order that, of course, has a maximal/minimal element. Let us consider the linear order of base types and the constraint graph that are shown in Figure 20. Clearly, $\theta = \{\gamma \mapsto \mathbb{F}, \beta \mapsto \mathbb{E}, \alpha \mapsto \mathbb{D}\}$ is a solution of the constraint graph. Now, assume $\gamma$ is the variable that is assigned first. It is assigned the type $\bigsqcup^{\{\beta::,\alpha::\mathcal{T}\}}_{\mathcal{U}} \{\mathbb{A}\}$, which happens to be $\mathbb{C}$. Unfortunately, after this assignment the constraint graph becomes unsolvable.

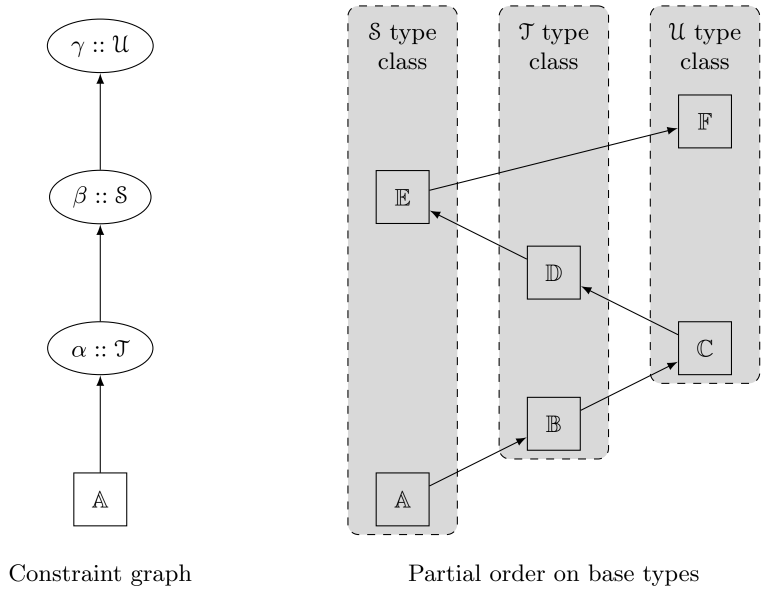


**Fig. 20.** Problematic example with a linear order subtype relation

There is no stronger restriction of the base type poset than a disjoint union of linear orders. Even with this restriction our algorithm is not complete for arbitrary sort quasi-orders. However, we have seen that our algorithm is complete if no type classes (i.e. only the type-class $\top$) are involved. Further, in case of a sort respecting poset, the algorithm shows a satisfying behaviour in practice even on input terms that involve type classes.

## 7 Related work

Type inference with automatic insertion of coercions in the context of functional programming languages was first studied by Mitchell [8,9]. First algorithms for type inference with subtypes were described by Fuh and Mishra [5] as well as Wand and O'Keefe [19]. The algorithm for constraint simplification presented in this paper resembles the MATCH algorithm by Fuh and Mishra. However, in order to avoid nontermination due to cyclic substitutions, they build up an extra data structure representing equivalence classes of atomic types, whereas we use a weak unification check suggested by Bourdoncle and Merz [2]. The seemingly simple problem of solving atomic subtype constraints has also been the subject of extensive studies. In their paper [5], Fuh and Mishra also describe a second algorithm CONSISTENT for solving this problem, but they do not mention any conditions for the subtype order on atomic types, so it is unclear whether their algorithm works in general. Simonet [17] presents general subtype constraint solvers and simplifiers for lattices designed for practical efficiency. Benke [1], as well as Pratt and Tiuryn [14] study the complexity of solving atomic constraints for a variety of different subtype orders. Extensions of Haskell with subtyping have been studied by Shields and Peyton Jones [16], as well as Nordlander [12].

## 7.1 Conclusion

Let us close with a few remarks on the realization of our algorithm in Isabelle. The abstract algorithm returns a set of results because coercion inference is ambiguous. For example, the term $sin(n{+}n)$, where $+ : \alpha \to \alpha \to \alpha$, $sin : \mathbb{R} \to \mathbb{R}$ and $n : \mathbb{N}$ has two type-correct completions with the coercion $real : \mathbb{N} \to \mathbb{R}$: $sin(real(n + n))$ and $sin(real\ n + real\ n)$. Our deterministic implementation happens to produce the first one. If the user wanted the second term, he would have to insert at least one *real* coercion. Because Isabelle is a theorem prover and because we did not modify its kernel, we do not have to worry whether the two terms are equivalent (this is known as coherence): in the worst case the system picks the wrong term and the proof one is currently engaged in fails or proves a different theorem, but it will still be a theorem.

To assess the effectiveness of our algorithm, we picked a representative Isabelle theory from real analysis (written at the time when all coercions had to be present) and removed as many coercions from it as our algorithm would allow — remember that some coercions may be needed to resolve ambiguity. Of 1061 coercions, only 221 remained. In contrast, the on-the-fly algorithm by Saïbi and Luo (see the beginning of §3) still needs 666 coercions.

We have not mentioned *let* so far because it does not mesh well with coercive subtyping. Consider the term $t = let\ f = s\ in\ u$ where $u = (Suc(f(0)), f(0.0))$, $0 : \mathbb{N}$, $Suc : \mathbb{N} \to \mathbb{N}$, $0.0 : \mathbb{R}$, and $s$ is a term that has type $\alpha \to \alpha$ under the constraints $\{\alpha \prec: \mathbb{R},\ \alpha \prec: \beta,\ \mathbb{N} \prec: \beta\}$. For example $s = \lambda x.\ if\, x = 0 \wedge sin(x) = 0.0\ then\ x\ else\ x)$ where $= : \alpha \to \alpha \to \mathbb{B}$ and $sin : \mathbb{R} \to \mathbb{R}$. Constraint resolution can produce the two substitutions $\{\alpha \mapsto \mathbb{N}, \beta \mapsto \mathbb{N}\}$ and $\{\alpha \mapsto \mathbb{R}, \beta \mapsto \mathbb{R}\}$, i.e. $s$ can receive the two types $\mathbb{N} \to \mathbb{N}$ and $\mathbb{R} \to \mathbb{R}$. In the natural extension of our algorithm we would use one of these types to type $u$ — which fails. However, if we consider $u[s/f]$ instead of $t$, our algorithm can insert suitable coercions to make the term type correct. Unfortunately this is not a shortcoming of the hypothetical extension of our algorithm but of coercive subtyping in general: There is no way to insert coercions into $t$ to make it type correct according to Hindley-Milner. If you want subtyping without extending the Hindley-Milner type system, this is where you pay the price.

## References

1. Benke, M.: Complexity of type reconstruction in programming languages with subtyping. Ph.D. thesis, Warsaw University (1997)
2. Bourdoncle, F., Merz, S.: On the integration of functional programming, class-based object-oriented programming, and multi-methods. Research Report 26, Centre de Mathématiques Appliquées, Ecole des Mines de Paris (Mar 1996)

3. Coq development team: The Coq proof assistant reference manual. INRIA (2010), `http://coq.inria.fr`, version 8.3
4. Frey, A.: Satisfying subtype inequalities in polynomial space. Theor. Comput. Sci. 277(1-2), 105–117 (2002)
5. Fuh, Y.C., Mishra, P.: Type inference with subtypes. In: ESOP, LNCS 300. pp. 94–114 (1988)
6. Luo, Z.: Coercions in a polymorphic type system. Mathematical Structures in Computer Science 18(4), 729–751 (2008)
7. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. 17(3), 348–375 (1978)
8. Mitchell, J.C.: Coercion and type inference. In: POPL. pp. 175–185 (1984)
9. Mitchell, J.C.: Type inference with simple subtypes. J. Funct. Program. 1(3), 245–285 (1991)
10. Nipkow, T.: Order-sorted polymorphism in Isabelle. In: Huet, G., Plotkin, G. (eds.) Logical Environments. pp. 164–188. CUP (1993)
11. Nipkow, T., Prehofer, C.: Type reconstruction for type classes. Journal of Functional Programming 5(2), 201–224 (1995)
12. Nordlander, J.: Polymorphic subtyping in O'Haskell. Sci. Comput. Program. 43(2-3), 93–127 (2002)
13. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)
14. Pratt, V.R., Tiuryn, J.: Satisfiability of inequalities in a poset. Fundam. Inform. 28(1-2), 165–182 (1996)
15. Saïbi, A.: Typing algorithm in type theory with inheritance. In: POPL. pp. 292–301 (1997)
16. Shields, M., Peyton Jones, S.: Object-oriented style overloading for Haskell. In: First Workshop on Multi-language Inferastructure and Interoperability (BABEL'01), Firenze, Italy (Sep 2001)
17. Simonet, V.: Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In: APLAS, LNCS 2895. pp. 283–302 (2003)
18. Tiuryn, J.: Subtype inequalities. In: LICS. pp. 308–315 (1992)
19. Wand, M., O'Keefe, P.: On the complexity of type inference with coercion. In: FPCA '89: Functional programming languages and computer architecture. pp. 293–298. ACM, New York, NY, USA (1989)
20. Wenzel, M.: Type classes and overloading in higher-order logic. In: TPHOLs. pp. 307–322 (1997)