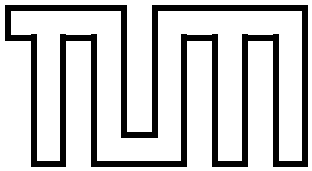


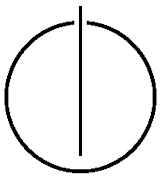
FAKULTÄT FÜR INFORMATIK

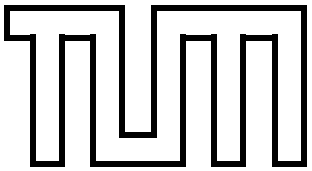
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

# Extension of a type inference algorithm with coercive subtyping

Dmytro Traytel





FAKULTÄT FÜR INFORMATIK

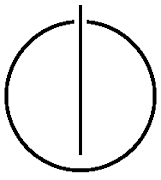
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

# Extension of a type inference algorithm with coercive subtyping

# Erweiterung eines Typinferenzalgorithmus um Subtyping mit Coercions


| | |
|---|---|
| Author: | Dmytro Traytel |
| Supervisor: | Prof. Tobias Nipkow, Ph.D. |
| Advisor: | Dr. Stefan Berghofer |
| Date: | May 14, 2010 |

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Munich, May 14, 2010 Dmytro Traytel

# Acknowledgments

I want to thank my supervisor Professor Nipkow for giving me the opportunity to work on a theoretical topic with an useful practical application. A special thanks goes to Dr. Stefan Berghofer, my advisor, who was never tired of helping me by contributing ideas and answering my questions. Without Stefan's deep understanding of the Isabelle system, the design and implementation of the presented extension would not have been possible.

I would also like to express my gratitude to all contributers of the "Isabelle Documentation Project" managed by Dr. Christian Urban. The draft of "The Isabelle Programming Tutorial" that emerged from this project really helped me to get familiar with the developer's level of Isabelle.

Further I am much obliged to Daniel Bader, Nikolaus Demmel, Vincenz Doelle, Martin Raiber, Johannes Schamburger and Anna Stucky for proofreading my thesis and helping me to improve the language and the content. Last but not least, I thank my family for their continuous support during my studies.

## Abstract

Subtyping with coercion semantics allows a type inference system to correct some ill-typed programs by the automatic insertion of implicit type conversions at run-time. This simplifies programmer's life but has its price: the general typability problem for given base type subtype dependencies is NP-complete. Nevertheless, if the given coercions define an order on types with certain properties, the problem behaves in a sane way in terms of complexity.

This thesis presents an algorithm that can be used to extend Hindley-Milner type inference with coercive subtyping assuming a given partial order on base types. Especially, we discuss restrictions on the subtype dependencies that are necessary to achieve an efficient implementation. Examples of problems that occur if these restrictions are not met are given. The result of these considerations is that the algorithm is complete if the given base type poset is a disjoint union of lattices. Furthermore, the interaction of subtyping with type classes is addressed. The algorithm that is extended to deal with type classes requires even a stronger restriction to assure completeness.

An ML-implementation of the presented algorithm is used in the generic proof assistant Isabelle.



## Zusammenfassung

Ein um Subtyping erweitertes Typinferenzsystem kann ein nicht typkorrektes Programm reparieren, indem es implizite Typumwandlungsfunktionen, sogenannte Coercions, automatisch einfügt. Dies vereinfacht das Leben der Programmierer, hat aber auch seinen Preis: das allgemeine Problem der Typüberprüfung bei vorgegebenen Subtypbeziehungen zwischen Basistypen ist NP-vollständig. Komplexitätstechnisch beherrschen kann man das Problem nur, wenn die Subtypbeziehungen eine Ordnung mit gewissen Eigenschaften darstellen.

In dieser Bachelor-Thesis wird ein Algorithmus zur Erweiterung von der Hindley-Milner Typinferenz um Subtyping mit Coercions vorgestellt. Ein wichtiger Aspekt der Arbeit ist die Diskussion der für eine effiziente Implementierung notwendigen Einschränkungen der Suptyprelation. Festgehalten wird dabei das Ergebnis der Vollständigkeit für den Algorithmus, falls die partielle Ordnung von Subtypen eine disjunkte Vereinigung von Verbänden ist. Die Notwendigkeit dieser Einschränkung wird mit anschaulichen Beispielen belegt. Darüber hinaus wird die Interaktion von Subtyping mit Typklassen behandelt. Um die Vollständigkeit der Erweiterung, die zusätzlich zu Subtyping auch noch Typklassen verarbeitet, zu garantieren, wird eine noch stärkere Restriktion der Subtyprelation erörtert.

Eine ML-Implementierung der vorgestellten Erweiterung wird in dem generischen Beweisassistenten Isabelle verwendet.

# Contents

*Subtyping is a cross-cutting extension, interacting with most other language features in non-trivial ways.*

Benjamin C. Pierce [Pie02]

# 1. Introduction

## 1.1. Motivation

The main idea of subtype polymorphism, or simply subtyping, extension is to allow the programmer to omit type conversions, also called coercions. Experienced object-oriented programmers are used to this extension in form of inheritance. For example, in Java an object of a subclass can always be treated as an object of a superclass. This happens automatically – the programmer does not have to insert any type casts to indicate this.

In functional programming languages with static typing subtyping is not a common feature. The main reason for this is the increase of complexity of a type inference system with subtyping compared to the well known algorithm W. The advantages of subtyping are underestimated. Type conversions should never contribute to the semantics of a program. It would clearly end up in madness if a type conversion from natural numbers to integers would change the value of the converted term. So type conversions can be regarded as unnecessary ballast to make a code type-correct. Allowing to remove them, subtyping improves the readability of a program a lot.

This applies at least as much to the field of semi-automated theorem proving. One of the core features of the generic proof assistant Isabelle is the Isar proof language. The biggest achievement of Isar is that it provides a language for writing down theorems and proofs, which are readable and understandable for both machines and humans (with a mathematical background). Type conversions are only relevant for the machine. It is desirable not to have them in the theorem and proof texts. Instead the type inference system should insert them automatically to make the source code type correct.

We present a generic algorithm that extends Hindley-Milner type inference with subtyping. The algorithm is implemented and used in the Isabelle proof assistant.

## 1.2. Related Work

John C. Mitchell [Mit84], [Mit91] was the first to reason about automatic coercion insertion in functional programming.

The COQ proof assistant uses a coercive subtyping algorithm that has some annoying issues that are discussed in chapter 3. Luo and Kieling [KL03] show how a similar algorithm can be used in Hindley-Milner type systems.

The basic algorithm presented in chapter 4 combines the works of Fuh and Mishra [FM88] and Wand and O'Keefe [WO89].

Our subtype constraint simplification is based on the MATCH-algorithm presented by Fuh and Mishra. MATCH in some way blends the usual unification with constraint simplification. Our intention was to provide a clean cut between these logically disjoint procedures.

The core idea of our algorithm to solve the simplified constraint set was introduced by Wand and O'Keefe . Unfortunately the complexity analysis in [WO89] was wrong. Mitchell and Lincoln [LM92] improved the requirements on the partial subtype order to have linear time complexity for this algorithm. Tiuryn [Tiu92] showed that the satisfiability problem for subtype inequalities is solvable in $PTIME$ if the underlying order of base types is a disjoint union of lattices.

## 1.3. Structure

Chapter 2 explains the notation used in this thesis. Then in chapter 3, basics of subtype polymorphism are covered. Also, a description of arising problems in case of the usage of a more or less naive subtype inference algorithm is given. Chapter 4 presents our algorithm for subtyping, which is extended for usage in a system with type classes in chapter 5. In chapter 6 some completeness problems of our algorithm for arbitrary subtype relations are analyzed. Thereafter, we propose restrictions on the subtype relation that establish the completeness property. Finally, chapter 7 provides some interesting implementation aspects on the extension of the Isabelle type inference with subtyping and concludes the thesis with a prospect of possible further developments.

# 2. Notation and terminology

We want to introduce some notation and terminology that is frequently used in this thesis. We assume that the reader in some way is familiar with the basics of type theory, especially with simply typed lambda calculus and Hindley-Milner type inference. [Pie02] and [Mil78] provide a comprehensive introduction to this field.

Let us define the base language of terms for our type inference. We use a simply typed lambda calculus with constants that have a predefined type signature.

$$\begin{array}{rcll} \langle \texttt{term} \rangle & = & \texttt{x} & \text{variable} \\ & | & \texttt{c} & \text{constant} \\ & | & \lambda \texttt{x} : \langle \textit{type} \rangle \,.\, \langle \texttt{term} \rangle & \text{abstraction} \\ & | & \langle \texttt{term} \rangle \, \langle \texttt{term} \rangle & \text{application} \end{array}$$

Note that the `typewriter` font is used for terms. We use the common syntactic sugar for a sequence of lambda abstractions: $\lambda \texttt{x y z} \,.\, \texttt{t}$ replaces $\lambda \texttt{x} \,.\, \lambda \texttt{y} \,.\, \lambda \texttt{z} \,.\, \texttt{t}$. Our terms can have the following types:

$$\begin{array}{rcll} \langle \textit{type} \rangle & = & \alpha & \text{type variable} \\ & | & T & \text{base type} \\ & | & C \, \langle \textit{type} \rangle \, \ldots \, \langle \textit{type} \rangle & \text{constructed type} \end{array}$$

The number of arguments of a type constructor $C$ is called the arity of $C$. Our type constructors have always at least arity one. We distinguish between base types and constructed types in the definition because in our algorithm these cases are treated completely different. The function type is a special case of a binary type constructor. We use the common infix notation $\tau \to \sigma$ instead of $\to \; \tau \; \sigma$ in this case.

Further we stick to the following naming conventions:

- $\alpha, \beta, \ldots$ denote type variables.
- $S, T, U, \ldots$ denote arbitrary base types.
- $C, D, \ldots$ denote type constructors.
- $\mathbb{A}, \mathbb{B}, \mathbb{C}, \ldots$ denote concrete base types. In some examples $\mathbb{B}$ denotes the boolean type. $\mathbb{P}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ correspond to prime, natural, integer, rational, real and complex numbers, respectively.
- $\tau, \sigma, \ldots$ denote arbitrary types.

- $\mathfrak{S}, \mathfrak{T}, \ldots$ denote sorts which are introduced in chapter 5.

The notation $\Sigma(\mathtt{c}) = \tau$ means that the constant $\mathtt{c}$ has the signature $\tau$. Free variables may occur in a signature of a constant. We write $\mathtt{c}_{[\overline{\alpha}\mapsto\overline{\tau}]}$ for an instantiation of the free variables $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$[1] in $\mathtt{c}$ with some types $\overline{\tau} = (\tau_1, \ldots, \tau_n)$.

A Hindley-Milner type inference system uses unification as a basic instrument. We assume that an unification algorithm is provided by the system that we are extending. The unification produces a set of type substitutions. We use $\theta$ as notation for an arbitrary substitution set and $\alpha \mapsto \tau$ if we want to specialize that $\alpha$ is substituted by $\tau$. Type substitutions can also be applied to terms, since terms also contain type annotations on constants and abstractions. For the application of a substitution to a term we use two notations. An arbitrary substitution $\theta$ applied to the type $\tau$ is denoted by $\theta\tau$. If we want to apply a concrete substitution $\alpha \mapsto \tau$ to the type $\sigma$, we write $\sigma[\alpha \mapsto \tau]$. Moreover, we use the same notation if we want to apply a substitution to any structure that contains terms or types, e.g. the constraint graph which is introduced in chapter 4.

Our notation for the subtyping relation is "$<:$". $\sigma <: \tau$ means $\sigma$ is a subtype of $\tau$. We write $\sigma <:_{\mathtt{f}} \tau$ to indicate that any term of type $\sigma$ can be transformed in a term of type $\tau$ by application of the function $\mathtt{f}$ of type $\sigma \to \tau$. $\mathtt{f}$ is called coercion. The coercion semantics of the subtype relation are covered in chapter 3. The subtyping relation is defined by a partial order on base types. In some examples we will use a Hasse diagram as a graphical representation of this partial order. In the diagram we place supertypes above subtypes. Thus, the partial order $\{\mathbb{N} <: \mathbb{R}\}$ is represented by figure 2.1.


Figure 2.1.: Simple partial order

We will also use the same representation for a set of subtype constraints between base types and type variables. Base types are denoted by rectangular nodes, type variables by round nodes and arbitrary types by octagonal nodes.

[1] The vector of free variables $\overline{\alpha}$ is ordered in a canonical way

# 3. Short introduction to subtyping

## 3.1. Coercion semantics of subtyping

There are two ways of interpreting the subtype relation. The first is called "subset semantics". It means that $S$ is a subtype of $T$ if the set of terms of type $S$ is a subset of the set of terms of type $T$. This provides a nice idea of what subtyping is, but is really unhandy to work with in a concrete implementation. The second possible semantics states that a type $S$ is a subtype of another type $T$ if there is a function of type $S \to T$ that is explicitly declared as a coercion. This "coercion semantics" is our way of understanding what defines a subtype.

We allow to declare only functions of type $S \to T$ where $S$ and $T$ are base types as coercions. Subtype relations between constructed types can be derived using map functions corresponding to the constructor.

**Definition 3.1** (Map function)**.** *Let $C$ be an n-ary type constructor. Then we call a function* `f` *of type*

$$f_1 \to f_2 \to \ldots \to f_n \to C\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n \to C\ \beta_1\ \beta_2\ \ldots\ \beta_n$$

*where $f_i$ has either the type $\alpha_i \to \beta_i$ or $\beta_i \to \alpha_i$ a **map function** for $C$. If $f_i$ has the type $\alpha_i \to \beta_i$ then $C$ is **covariant** in the i-th argument w.r.t.* `f`*. Otherwise $C$ is **contravariant** in the i-th argument w.r.t.* `f`*.*

Of course, there are different functions that are map functions for a single type constructor. With "reasonable" map functions we refer to the map functions that really apply all $f_i$ to all subterms of the corresponding type $\tau_i$ or $\sigma_i$.

**Lemma 3.2** (Subtype relation for constructed types)**.** *Let $C$ be an n-ary type constructor and* `f` *a reasonable map function for $C$. Then it holds:*

$$\forall i = 1 \ldots n : \begin{cases} \tau_i <:_{\mathtt{f}_i} \sigma_i & \text{when } C \text{ covariant in the i-th argument w.r.t. } \mathtt{f} \\ \sigma_i <:_{\mathtt{f}_i} \tau_i & \text{when } C \text{ contravariant in the i-th argument w.r.t. } \mathtt{f} \end{cases}$$

$$\Leftrightarrow$$

$$C\ \tau_1\ \tau_2\ \ldots\ \tau_n <:_{\mathtt{f}\ \mathtt{f}_1\ \mathtt{f}_2\ \ldots\ \mathtt{f}_n} C\ \sigma_1\ \sigma_2\ \ldots\ \sigma_n$$

*Proof.* "⇒": The function `f f`$_1$ `f`$_2$ $\ldots$ `f`$_n$ converts terms of type $C\ \tau_1\ \tau_2\ \ldots\ \tau_n$ in terms of type $C\ \sigma_1\ \sigma_2\ \ldots\ \sigma_n$.

"⇐": By definition of a map function.

□

For example, if $S <:_{\texttt{T\_of\_S}} T$ then $List\ S <:_{\texttt{map T\_of\_S}} List\ T$ where map: $(\alpha \to \beta) \to List\ \alpha \to List\ \beta$ is the list map function.

An other interesting example is the function type which can be regarded as a binary type constructor. The function type is contravariant in the first argument. An explanation for this is given in [Pie02]. A reasonable function type map function is the following: `fun_map = λ f1 f2 f x . f2 (f (f1 x))`. The type of `fun_map` is $(\gamma \to \alpha) \to (\beta \to \delta) \to (\alpha \to \beta) \to (\gamma \to \delta)$. Thus from $T_1 <:_{\texttt{S1\_of\_T1}} S_1$ and $S_2 <:_{\texttt{T2\_of\_S2}} T_2$ we can derive $S_1 \to S_2 <:_{\texttt{fun\_map S1\_of\_T1 T2\_of\_S2}} T_1 \to T_2$.

## 3.2. Subsumption rule

The theoretical extension of a Hindley-Milner type inference system with subtyping is easy. It is done by adding a single, so called "subsumption" rule to the common inference rules. The judgement $\Gamma \vdash \texttt{t} \rightsquigarrow \texttt{u} : T$ means that the term t gets transformed by coercion insertion to the term u that has the type $T$ in the context $\Gamma$.

$$\frac{\Gamma \vdash \texttt{t} \rightsquigarrow \texttt{u} : S \qquad S <:_{\texttt{c}} T}{\Gamma \vdash \texttt{t} \rightsquigarrow \texttt{c u} : T}\ \textsc{Subsumption}$$

Unfortunately, this works that easy only in theory. The rules of a type inference system including this subsumption rule are not syntax directed. That means that if we proceed as we do during type inference, reading the rules from bottom to top, more than one rule can be applicable. Actually the subsumption rule is always applicable, so we have to guess where to apply it.

However, this problem can be avoided by rewriting the derivation trees of the type inference system with subsumption. Executing this, Pierce [Pie02] shows that the only rule where subsumption must be used in a syntax directed approach is the application rule.

$$\frac{\Gamma \vdash \texttt{t}_1 \rightsquigarrow \texttt{u}_1 : T_{11} \to T_{12} \qquad \Gamma \vdash \texttt{t}_2 \rightsquigarrow \texttt{u}_2 : T_2 \qquad T_2 <:_{\texttt{c}} T_{11}}{\Gamma \vdash \texttt{t}_1\ \texttt{t}_2 \rightsquigarrow \texttt{u}_1\ (\texttt{c}\ \texttt{u}_2) : T_{12}}\ \textsc{App}$$

## 3.3. Inserting coercions "on the fly"

The simplest extension solving the subtyping problem that seems reasonable at first glance is to try to insert a coercion immediately when a subtype relation $S <: T$ occurs in the typing derivation and to fail if no coercion from $S$ to $T$ exists. This way to handle subtyping is used in the Coq proof assistant. However, there are issues.

**Example 3.3.** *Let us consider the type reconstruction of the terms* `f 0 True` *and* `f True 0` *with the following type signatures:* $\Sigma(\texttt{f}) = \alpha \to \alpha \to \mathbb{B}$, $\Sigma(\texttt{True}) = \mathbb{B}$ *and* $\Sigma(\texttt{0}) = \mathbb{N}$. *Further* $\mathbb{B}$ *should be a subtype of* $\mathbb{N}$ *via coercion* nob: $\mathbb{B} \to \mathbb{N}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma(\mathtt{f}) = \alpha \to \alpha \to \mathbb{B}}{\Gamma \vdash \mathtt{f} \rightsquigarrow \mathtt{f} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}} \quad \dfrac{\Sigma(\mathtt{0}) = \mathbb{N}}{\Gamma \vdash \mathtt{0} \rightsquigarrow \mathtt{0} : \mathbb{N}} \quad \dfrac{}{\mathbb{N} <:_{\mathtt{id}} \mathbb{N}}}{\Gamma \vdash \mathtt{f\ 0} \rightsquigarrow \mathtt{f\ 0} : \mathbb{N} \to \mathbb{B}} \quad \dfrac{\dfrac{\Sigma(\mathtt{True}) = \mathbb{B}}{\Gamma \vdash \mathtt{True} \rightsquigarrow \mathtt{True} : \mathbb{B}} \quad \dfrac{}{\mathbb{B} <:_{\mathtt{nob}} \mathbb{N}}}{}}{\Gamma \vdash \mathtt{f\ 0\ True} \rightsquigarrow \mathtt{f\ 0\ (nob\ True)} : \mathbb{B}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma(\mathtt{f}) = \alpha \to \alpha \to \mathbb{B}}{\Gamma \vdash \mathtt{f} \rightsquigarrow \mathtt{f} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}} \quad \dfrac{\Sigma(\mathtt{True}) = \mathbb{B}}{\Gamma \vdash \mathtt{True} \rightsquigarrow \mathtt{True} : \mathbb{B}} \quad \dfrac{}{\mathbb{B} <:_{\mathtt{id}} \mathbb{B}}}{\Gamma \vdash \mathtt{f\ True} \rightsquigarrow \mathtt{f\ True} : \mathbb{B} \to \mathbb{B}} \quad \dfrac{\Sigma(\mathtt{0}) = \mathbb{N}}{\Gamma \vdash \mathtt{0} \rightsquigarrow \mathtt{0} : \mathbb{N}} \quad \dfrac{\lightning}{\mathbb{N} <:_{\mathtt{c}} \mathbb{B}}}{\Gamma \vdash \mathtt{f\ True\ 0} \rightsquigarrow \mathtt{f\ True\ (c\ 0)} : \mathbb{B}}$$

*We see that the immediate coercion insertion works with the first term but fails dealing with the second. Instead, the type of* `f True 0` *should be inferred as following:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma(\mathtt{f}) = \alpha \to \alpha \to \mathbb{B}}{\Gamma \vdash \mathtt{f} \rightsquigarrow \mathtt{f} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}} \quad \dfrac{\Sigma(\mathtt{True}) = \mathbb{B}}{\Gamma \vdash \mathtt{True} \rightsquigarrow \mathtt{True} : \mathbb{B}} \quad \dfrac{}{\mathbb{B} <:_{\mathtt{nob}} \mathbb{N}}}{\Gamma \vdash \mathtt{f\ True} \rightsquigarrow \mathtt{f\ (nob\ True)} : \mathbb{N} \to \mathbb{B}} \quad \dfrac{\Sigma(\mathtt{0}) = \mathbb{N}}{\Gamma \vdash \mathtt{0} \rightsquigarrow \mathtt{0} : \mathbb{N}} \quad \dfrac{}{\mathbb{N} <:_{\mathtt{id}} \mathbb{N}}}{\Gamma \vdash \mathtt{f\ True\ 0} \rightsquigarrow \mathtt{f\ (nob\ True)\ 0} : \mathbb{B}}$$

*So, we have to insert a coercion while we derive the type of* `f True`*, which is not possible using the described approach as* `f True` *is already type correct without any coercions.*

The Coq reference manual [dt06] claims that this is the "normal" behaviour of coercions. Still, almost all advantages of subtyping are lost if the programmer has to reason about the order of arguments. Due to these considerations, the goal is to provide an algorithm that treats such symmetric cases as shown above in the same way.

# 4. Coercive subtyping using subtype constraints

## 4.1. Overview

The approach presented here will not try to insert coercions at function applications where the argument type differs from the function domain immediately while inferring the type of the function application. Instead, we generate subtype constraints. The entirety of all constraints provides us a global view on the term that we are processing. Thus, we cannot run into the problems described at the end of the previous chapter where we are only locally checking subterms while inferring the type.

In the following, we assume a fixed set $\mathcal{M}$ of reasonable and unique map functions for type constructors and a partial order on base types given by a set of defined coercions $\mathcal{C}$. In chapter 6 we show that to assure completeness of our algorithm we need to restrict the order on base types to a disjoint union of lattices.

The algorithm can be divided in four major phases. A visualization of the main steps of the algorithm in form of a control flow can be found in the appendix A. First, we need to generate the subtype constraints traversing the term recursively. These constraints are inequalities on arbitrary types. Then, we simplify the constraints until the constraint set contains only inequalities between base types and variables. The next step is to organize these simplified inequalities in a graph and solve them. Solving means in this case to find a substitution. Applying a solving substitution to the whole constraint set results in inequalities that are consistent with the given partial order on base types. The terms solvability and satisfiability are used synonymously in this thesis. Finally, the coercions are inserted while traversing the term for the second time and applying the solving substitution to the subterms. Separating the insertion of coercions from the constraint processing is not necessary but allows not to care about how the coercions have to be transformed during constraint simplification.

## 4.2. Constraint preprocessing

### 4.2.1. Constraint generation

We generate constraints with the following inference rules. The judgement $\Gamma \vdash \mathtt{t} : T\nabla(U, S)$ means that the term $\mathtt{t}$ has the type $T$ constrained with the unification problems $U$ and the subtype inequalities $S$ in the context $\Gamma$. Our implementation of the SUBCT-CONST rule introduces fresh variables $\overline{\tau}$ that replace the polymorphic variables $\overline{\alpha}$ in the type signature $\Sigma(\mathtt{c})$ of a constant $\mathtt{c}$ in purpose to provide the Hindley-Milner polymorphism. A constant that is not annotated with such a substitution does not contain polymorphic variables.

CONSTRAINT GENERATION RULES

$$\frac{\mathtt{x} : T \in \Gamma}{\Gamma \vdash \mathtt{x} : T\nabla(\emptyset, \emptyset)}\ \textsc{SubCT-Var}$$

$$\frac{\Sigma(\mathtt{c}) = T}{\Gamma \vdash \mathtt{c}_{[\overline{\alpha}\mapsto\overline{\tau}]} : T[\overline{\alpha} \mapsto \overline{\tau}]\nabla(\emptyset, \emptyset)}\ \textsc{SubCT-Const}$$

$$\frac{\Gamma, \mathtt{x} : T_1 \vdash \mathtt{t} : T_2\nabla(U, S)}{\Gamma \vdash \lambda \mathtt{x} : T_1 \,.\, \mathtt{t} : T_1 \to T_2\nabla(U, S)}\ \textsc{SubCT-Abs}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : T_1\nabla(U_1, S_1) \qquad \Gamma \vdash \mathtt{t}_2 : T_2\nabla(U_2, S_2) \qquad \overbrace{\alpha, \beta}^{\text{fresh}}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \beta\nabla(U_1 \cup U_2 \cup \{T_1 \doteq \alpha \to \beta\}, S_1 \cup S_2 \cup \{T_2 <: \alpha\})}\ \textsc{SubCT-App}$$

**Example 4.1.** *Let us apply the rules to the failing example from the previous chapter:*

$$\dfrac{\dfrac{\dfrac{\Sigma(\mathtt{f}) : \alpha \to \alpha \to \mathbb{B}}{\Gamma \vdash \mathtt{f}_{[\alpha\mapsto\alpha_3]} : \alpha_3 \to \alpha_3 \to \mathbb{B}\nabla(\emptyset, \emptyset)} \quad \dfrac{\Sigma(\mathtt{True}) : \mathbb{B}}{\Gamma \vdash \mathtt{True} : \mathbb{B}\nabla(\emptyset, \emptyset)} \quad \overbrace{\alpha_2, \beta_2}^{\textit{fresh}}}{\Gamma \vdash \mathtt{f}_{[\alpha\mapsto\alpha_3]}\mathtt{True} : \beta_2\nabla(\{\alpha_3 \to \alpha_3 \to \mathbb{B} \doteq \alpha_2 \to \beta_2\}, \{\mathbb{B} <: \alpha_2\})} \quad \dfrac{\Sigma(\mathtt{0}) : \mathbb{N}}{\Gamma \vdash \mathtt{0} : \mathbb{N}\nabla(\emptyset, \emptyset)} \quad \overbrace{\alpha_1, \beta_1}^{\textit{fresh}}}{\Gamma \vdash \mathtt{f}_{[\alpha\mapsto\alpha_3]}\mathtt{True}\ \mathtt{0} : \beta_1\nabla(\{\alpha_3 \to \alpha_3 \to \mathbb{B} \doteq \alpha_2 \to \beta_2, \beta_2 \doteq \alpha_1 \to \beta_1\}, \{\mathbb{B} <: \alpha_2, \mathbb{N} <: \alpha_1\})}$$

The unification constraints $U$ can be solved immediately. The resulting substitution $\theta_{\text{init}}$ is applied to the subtypes constraints $S$.

**Example 4.2** (example 4.1 continued)**.** *In our example we obtain the substitution $\theta_{init} = \{\alpha_2 \mapsto \alpha_3, \beta_2 \mapsto \alpha_3 \to \mathbb{B}, \alpha_1 \mapsto \alpha_3, \beta_1 \mapsto \mathbb{B}\}$ and apply it to the constraint set $\theta_{init}S = \{\mathbb{B} <: \alpha_3, \mathbb{N} <: \alpha_3\}$.*

### 4.2.2. Constraint simplification

**Definition 4.3** (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):*

- $\alpha <: \beta$
- $\alpha <: T$
- $T <: \alpha$

The generated constraints are inequalities between arbitrary types. Our goal is to have only atomic constraints. We can achieve this by applying the following rules to the constraint set repeatedly.

CONSTRAINT SIMPLIFICATION RULES

1. Transfer $C\ \tau_1\ \tau_2\ \ldots\ \tau_n <: C\ \sigma_1\ \sigma_2\ \ldots\ \sigma_n$ to $n$ new subtype constraints or unification problems:

$$\forall i = 1 \ldots n \begin{cases} \tau_i <: \sigma_i & C \text{ is covariant in the i-th argument} \\ & \text{w.r.t. to the known map function for } C \\ \sigma_i <: \tau_i & C \text{ is contravariant in the i-th argument} \\ & \text{w.r.t. to the known map function for } C \\ \tau_i \doteq \sigma_i & \text{no map function known for } C \end{cases}$$

   If unification constraints are generated, solve them directly and apply the resulting substitution to the whole constraint set.

2. Transfer $\alpha <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n$ to $C\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n$ using fresh variables $\alpha_1, \ldots, \alpha_n$. Apply the substitution $\{\alpha \mapsto C\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n\}$ to the whole constraint set.

   Same procedure for the symmetric constraint $C\ \tau_1\ \tau_2\ \ldots\ \tau_n <: \alpha$.

3. Remove $S <: T$ for base types $S$ and $T$ if $S$ is a subtype of $T$[2]. Fail otherwise.

4. Fail on any non-atomic constraint that is not matched by rules 1-3.

If none of the rules is applicable, there are only atomic constraints left. An interesting question is whether this 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. To eliminate such cases it suffices to check whether for every constraint $\tau <: \sigma$, $\tau$ is unifiable with $\sigma$. Of course, what we need here is a weaker notion of unification, since we are unifying constraints where different base types may be compared.

**Definition 4.4** (Weak unification)**.** *Two types are **weakly unifiable** if they are unifiable regarding all base types as equal. We call a substitution that is a most general unifier of these types regarding all base types as equal a **weak substitution**.*

So, before starting to apply the simplification rules on the constraint set, we check whether all constraints are weakly unifiable. This test makes rule 4 obsolete since only non-atomic constraints that match one of the rules 1-3 pass the weak unification test.

**Theorem 4.5** (Termination)**.** *The subtype simplification will terminate if the weak unification test on the constraint set passes.*

*Proof (based on the proof of lemma 13 in [BM96]).* 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 $i$, $\theta_i$ a weak substitution for $S_i$. If we apply rule 1 or rule 3, $\theta_i$ is still a weak substitution for the resulting constraint set. The application of rule 2 to the subtype constraint $\alpha <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n$ introduces new variables $\alpha_1, \ldots, \alpha_n$ and applies the substitution

[2] $S$ and $T$ might be the same type here

$\langle \alpha := C\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n \rangle$ to the whole constraint set. Since $\{\alpha <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n\} \in S_i$ and $S_i$ is weakly unifiable, $\theta_i\alpha$ must be a type constructed by $C$ so that replacing $\alpha$ with $C\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n$ will not produce a clash of different type constructors. Thus, $S_{i+1}$ is still weakly unifiable. It is part of the algorithm to test whether the starting constraint set is weakly unifiable. We will only start to simplify the constraints if the test passes. By induction, we obtain weak unifiability for every subtype constraint set produced by our simplification algorithm.

Now, we show the termination of the constraint simplification by specifying a measure that decreases with every non-failing application of any rule. Let $\#(\tau)$ be the following recursively defined number:

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

Further, we define two measures for a weakly unifiable subtype constraint set $S$:

$$|S| = \sum_{\{\tau <: \sigma\} \in S} \#(\tau) + \#(\sigma) \text{ and } |S|_{TV} = \sum_{x \in TV(S)} \#(\theta x)$$

where $\theta$ is a most general unifier for $S$ and $TV(S)$ denotes the type variables occurring in $S$.

Consider the lexicographic order on tuples $(|S_i|_{TV}, |S_i|)$ for the coercion sets $S_i$ in any step $i$. If rule 1 in case of a known map function or rule 3 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 $\theta_{i+1}$ (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 rule 2 is applied to the constraint $\alpha <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n$, we notice that the weak substitutions $\theta_i$ and $\theta_{i+1}$ only differ concerning the variables $\alpha$ and $\alpha_1, \ldots, \alpha_n$. Further, $\theta_i$ and $\theta_{i+1}$ are associated with the following equality: $\theta_i\alpha = \theta_{i+1}(C\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n)$. Now, we can conclude

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

which together with the fact $TV(S_{i+1}) = (TV(S_i) \setminus \{\alpha\}) \cup \{\alpha_1, \ldots, \alpha_n\}$ implies $|S_i|_{TV} > |S_{i+1}|_{TV}$.

If rule 1 in case of an unknown map function 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 $|S_i|_{TV} = |S_{i+1}|_{TV}$. Further, $|S_i| > |S_{i+1}|$ holds because a non-atomic 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. □

Beside producing the atomic constraint set, the constraint simplification should also remember the substitutions applied in rule 1 and rule 2 storing them in $\theta_{\text{init}}$. We refer to the resulting substitu-

tion as $\theta_{\text{simp}}$. Note that the weak unification test also produces a substitution which is **not** contained in $\theta_{\text{simp}}$.

**Theorem 4.6** (Consistency)**.** *Let $\theta_{sol}$ be a substitution that solves the atomic constraint set produced by constraint unification. Then $\theta_{simp} \cup \theta_{sol}$ solves the original constraint set.*

*Intuition.* The constraint simplification decomposes the dependencies between constructed types into atomic constraints correctly using lemma 3.1. The information about the dependencies between variables and constructed types, which is not contained in the atomic constraint set, is stored in $\theta_{\text{simp}}$. □

## 4.3. 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 way of thinking allows us to speak of predecessors and successors of a type. In our algorithm we will use the transitive closure of the constraint graph. The transitive arcs can be stored directly in the graph or be computed every time they are needed. In a concrete implementation a trade-off between these two possibilities is required.

### 4.3.1. Graph construction

Building such a graph is straightforward. The only thing one has to care about is the handling of 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[3], the inference will fail. Unifiable cycles should be eliminated with the iterated application of the following procedure in which $G = (V, E)$ is the constraint graph and $K$ is the set of nodes of the cycle.

CYCLE ELIMINATION ALGORITHM (A SINGLE ITERATION)

1. $P = \{x \mid \exists c \in K : (x, c) \in E\} \setminus K$
2. $S = \{x \mid \exists c \in K : (c, x) \in E\} \setminus K$
3. Let $\tau$ be the most general unifier of $K$ applied to any type in $K$
4. $(V', E') = G[V \setminus K]$ where the notation $G[S]$ describes the subgraph of $G$ induced by $S$
5. Update $G \leftarrow (V' \cup \{\tau\}, E' \cup \{(x, \tau) \mid x \in P\} \cup \{(\tau, x) \mid x \in S\})$
6. Update the substitution $\theta_{\text{simp}} \leftarrow \theta_{\text{simp}} \cup \{c \mapsto \tau \mid c \in K\}$

[3] From this point on, the standard unification is used again, not the weak unification.

**Lemma 4.7** (Termination). *The cycle elimination terminates.*

*Proof.* Obviously, since in the non-failing case $|E| + |V|$ decreases after each iteration. □

Figure 4.1 visualizes the procedure handling an example graph containing a cycle.

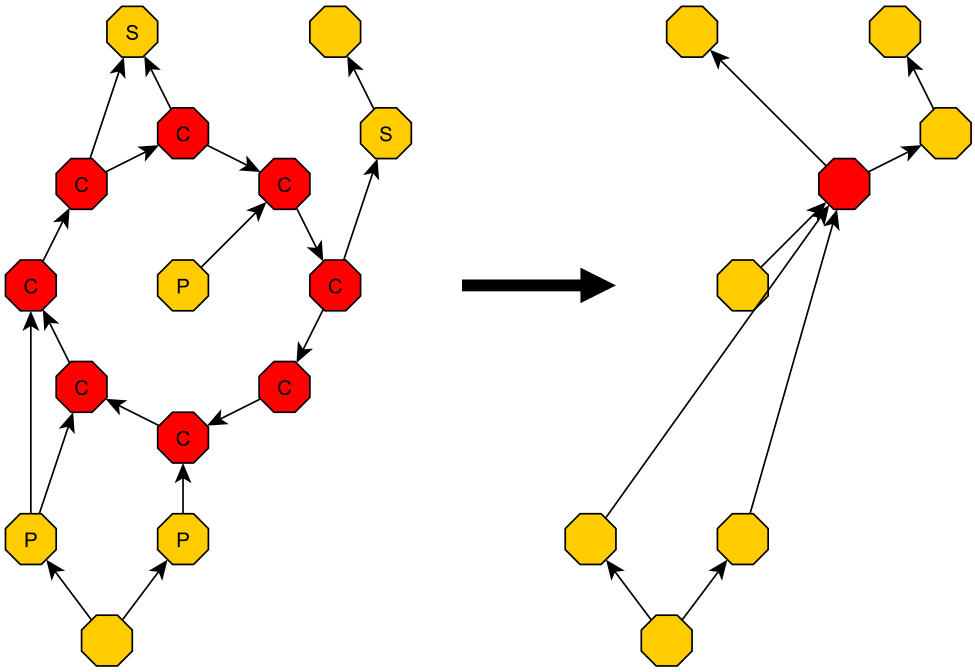



Figure 4.1.: Collapse of a cycle in a graph

We call the substitution obtained from cycle elimination $\theta_{\text{cyc}}$.

### 4.3.2. Constraint resolution

Now, we must find an assignment for all variables that appear in the graph $G$. We use an algorithm that is based on the approach presented in [WO89].

First, we define a notation for supertypes, subtypes, greatest lower (infimum) and least upper (supremum) bounds of two or a set of base types.

**Definition 4.8** (Notation). *Let $S, T$ be base types and $X$ a set of base types. We define[4] w.r.t. the given subtype relation "$<:$":*

- $\overline{T} = \{T' \mid T <: T', T' \text{ base type}\}$ *the set of* ***supertypes***
- $\underline{T} = \{T' \mid T' <: T, T' \text{ base type}\}$ *the set of* ***subtypes***
- ***Supremum*** *of $S$ and $T$: $T \sqcup S \in \overline{T} \cap \overline{S}$ and $\forall U \in \overline{T} \cap \overline{S} : T \sqcup S <: U$*
- ***Infimum*** *of $S$ and $T$: $T \sqcap S \in \underline{T} \cap \underline{S}$ and $\forall L \in \underline{T} \cap \underline{S} : L <: T \sqcap S$*

[4]This definition includes some procedural semantics, i.e. the non-existence of a type that satisfies the supremum/infimum property produces a failure during computation.

- ***Supremum*** *of* $X$: $\bigsqcup X \in \bigcap_{T \in X} \overline{T}$ *and* $\forall U \in \bigcap_{T \in X} \overline{T} : \bigsqcup X <: U$
- ***Infimum*** *of* $X$: $\bigsqcap X \in \bigcap_{T \in X} \underline{T}$ *and* $\forall L \in \bigcap_{T \in X} \underline{T} : L <: \bigsqcap X$

The algorithm iterates the following two loops until no new variable is assigned during a whole iteration.

CONSTRAINT RESOLUTION ALGORITHM (A SINGLE ITERATION)

1. For every unassigned variable $\alpha$ in $G$ do
   a) Let $P_\alpha$ be the set of all base type predecessors[5] of $\alpha$ in $G$
   b) If $P_\alpha = \emptyset$, skip steps c)-e)
   c) Let $S_\alpha$ be the set of all base type successors[5] of $\alpha$ in $G$
   d) If $\bigsqcup P_\alpha$ exists and $\forall$ base types $\beta \in S_\alpha : \bigsqcup P_\alpha <: \beta$, update $\theta_{\text{cyc}} \leftarrow \theta_{\text{cyc}} \cup \{\alpha \mapsto \bigsqcup P_\alpha\}$ and $G \leftarrow \theta_{\text{cyc}} G$
   e) Fail otherwise.
2. For every unassigned variable $\alpha$ in $G$ do
   a) Let $S_\alpha$ be the set of all base type successors[5] of $\alpha$ in $G$
   b) If $S_\alpha = \emptyset$, skip steps c)-d)
   c) If $\bigsqcap S_\alpha$ exists, update $\theta_{\text{cyc}} \leftarrow \theta_{\text{cyc}} \cup \{\alpha \mapsto \bigsqcap P_\alpha\}$ and $G \leftarrow \theta_{\text{cyc}} G$
   d) Fail otherwise.

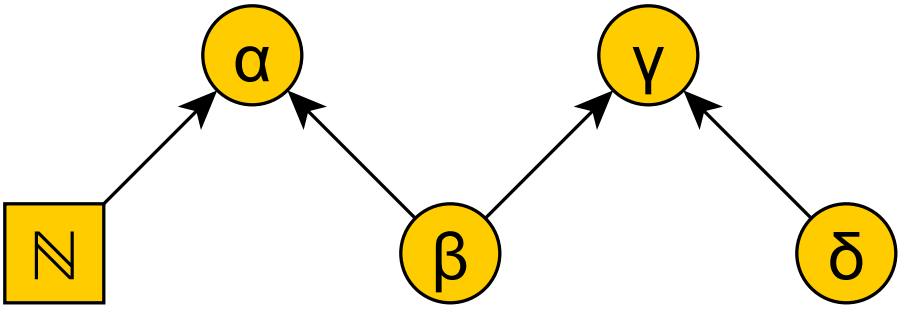



Figure 4.2.: Crown-shaped constraint graph

The original algorithm of Wand an O'Keefe does not iterate these two loops. However, this is necessary to resolve so called "crowns". Figure 4.2 visualizes this problematic case which, e.g. occurs as subgraphs of the constraint graph during the type inference of the term `g (f 1 x) (f x y)`. The signatures of the constants in this example are the following:

[5] This includes also transitive arcs.

- $\Sigma(\texttt{g}) = \mathbb{B} \to \mathbb{B} \to \mathbb{B}$
- $\Sigma(\texttt{f}) = \alpha \to \alpha \to \mathbb{B}$
- $\Sigma(\texttt{1}) = \mathbb{N}$
- $\Sigma(\texttt{x}) = \alpha$
- $\Sigma(\texttt{y}) = \alpha$.

After a single iteration is applied to such a crown not all variables will be assigned. The further iterations ensure that all subtype dependencies are respected.

FINAL STEP OF THE CONSTRAINT RESOLUTION

By construction, unassigned variables 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 produces some new substitutions. These are added to the substitution $\theta_{\text{cyc}}$ which results in a substitution that is called $\theta_{\text{sol}}$.

**Example 4.9** (example 4.2 continued)**.** *Our running example passes constraint simplification and building of the constraint graph without further modification of the constraints. We obtain the constraint graph shown in figure 4.3.*

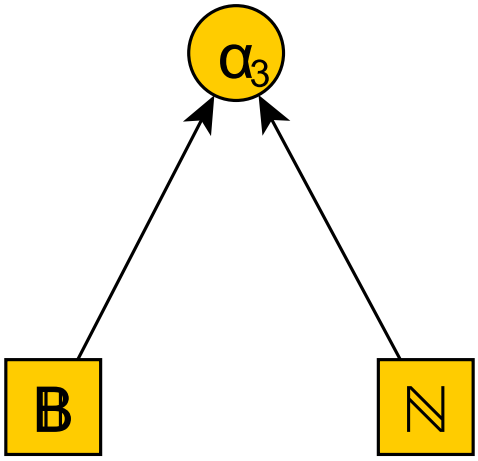



Figure 4.3.: Constraint graph of `f True 0`

*The constraint resolution algorithm assigns $\alpha_3$ the least upper bound of $\{\mathbb{B}, \mathbb{N}\}$ which of course is $\mathbb{N}$. The resulting substitution is $\theta_{sol} = \{\alpha_2 \mapsto \alpha_3, \beta_2 \mapsto \alpha_3 \to \mathbb{B}, \alpha_1 \mapsto \alpha_3, \beta_1 \mapsto \mathbb{B}, \alpha_3 \mapsto \mathbb{N}\}$*

**Lemma 4.10** (Termination)**.** *The constraint resolution terminates.*

*Proof.* Obviously, since either the algorithm terminates or the number of unassigned variables decreases after every iteration. □

**Theorem 4.11** (Consistency)**.** *For every arc $(\tau, \sigma)$ of $\theta_{sol}G$ it holds either $\tau$ and $\sigma$ are both the same type variable or $\tau$ and $\sigma$ are both base types and $\tau <: \sigma$ holds w.r.t. the given order on base types.*

*Intuition.* The case of $\tau$ and $\sigma$ being the same type variable is obvious. Every assignment done in loop 1 by the algorithm takes all successors and predecessors of the assigned variable into account. In loop 2 we do not have to check the predecessors, since a variable that is unassigned after the first loop does not have any base type predecessors. □

Unfortunately this algorithm will not solve all satisfiable constraint sets for arbitrary posets on base types. In chapter 6 we will provide examples of posets and subtype constraints that can not be inferred. Restricting the partial order to be a disjoint union of lattices suffices to make the algorithm complete.

## 4.4. Coercion insertion

Finally, we have a solving substitution $\theta_{\text{sol}}$. Applying this substitution to the term annotated with instantiations for polymorphic variables, which we obtain from the constraint generation step, will produce a term that can always be coerced to a type correct term.

### 4.4.1. Insertion rules

We insert coercions at every function application using the following inference rules. The judgement $\Gamma \vdash \mathtt{t} \rightsquigarrow \mathtt{u} : T$ means that in context $\Gamma$ the possibly ill-typed term $t$ gets transformed by coercion insertion and type substitution on constants and abstractions to a well-typed term $u$ of type $T$.

COERCION INSERION RULES

$$\frac{\mathtt{x} : T \in \Gamma}{\Gamma \vdash \mathtt{x} \rightsquigarrow \mathtt{x} : T} \text{ COERCE-VAR}$$

$$\frac{\Sigma(\mathtt{c}) = T}{\Gamma \vdash \mathtt{c}_{[\overline{\alpha} \mapsto \overline{\tau}]} \rightsquigarrow \theta_{\text{sol}}\mathtt{c}_{[\overline{\alpha} \mapsto \overline{\tau}]} : \theta_{\text{sol}} T[\overline{\alpha} \mapsto \overline{\tau}]} \text{ COERCE-CONST}$$

$$\frac{\Gamma, \mathtt{x} : \theta_{\text{sol}} T_1 \vdash \mathtt{t} \rightsquigarrow \mathtt{u} : T_2}{\Gamma \vdash \lambda \mathtt{x} : T_1 \,.\, \mathtt{t} \rightsquigarrow \lambda \mathtt{x} : \theta_{\text{sol}} T_1 \,.\, \mathtt{u} : \theta_{\text{sol}} T_1 \to T_2} \text{ COERCE-ABS}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 \rightsquigarrow \mathtt{u}_1 : T_{11} \to T_{12} \qquad \Gamma \vdash \mathtt{t}_2 \rightsquigarrow \mathtt{u}_2 : T_2 \qquad T_2 <:_{\mathtt{c}} T_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 \rightsquigarrow \mathtt{u}_1\ (\mathtt{c}\ \ \mathtt{u}_2) : T_{12}} \text{ COERCE-APP}$$

### 4.4.2. Generation rules

The coercion insertion rules do not specify what the coercions exactly are. Therefore, we provide the following four rules:

COERCION GENERATION RULES

$$\frac{}{T <:_{\mathtt{id}} T}\ \textsc{Refl}$$

$$\frac{\mathtt{c} : T \to S \in \mathcal{C}}{T <:_{\mathtt{c}} S}\ \textsc{Def}$$

$$\frac{\mathtt{c}_1 : T \to R \in \mathcal{C} \qquad \mathtt{c}_2 : R \to S \in \mathcal{C}}{T <:_{\lambda \mathtt{x}:\, T\,.\,\mathtt{c}_2\ (\mathtt{c}_1\ \mathtt{x})} S}\ \textsc{Trans}$$

$$\frac{\begin{array}{c}\mathtt{map}_C : \rho_1 \to \ldots \to \rho_n \to C\ \alpha_1 \ldots \alpha_n \to C\ \beta_1 \ldots \beta_n \in \mathcal{M} \qquad \theta_{\text{inst}} = \{\overline{\alpha} \mapsto \overline{\tau}, \overline{\beta} \mapsto \overline{\sigma}\} \\ \forall i = 1 \ldots n : \ \ \mathtt{c}_i : \theta_{\text{inst}}\rho_i \in \mathcal{C},\ \theta_{\text{inst}}\rho_i \in \{\tau_i \to \sigma_i, \sigma_i \to \tau_i\}\end{array}}{C\ \tau_1 \ldots \tau_n <:_{\theta_{\text{inst}}\mathtt{map}_C\ \mathtt{c}_1 \ldots \mathtt{c}_2}\ C\ \sigma_1 \ldots \sigma_n}\ \textsc{Cons}$$

CONS and DEF are given by definition of the coercion semantics. TRANS and REFL are justified by the fact that the subtype order is reflexive and transitive. Transitivity causes some ambiguity.

**Example 4.12.** *Consider the subtype order* $\{\mathbb{A} <:_{\mathtt{boa}} \mathbb{B}, \mathbb{A} <:_{\mathtt{coa}} \mathbb{C}, \mathbb{B} <:_{\mathtt{dob}} \mathbb{D}, \mathbb{C} <:_{\mathtt{doc}} \mathbb{D}\}$*. Now, there are two possible coercions from* $\mathbb{A}$ *to* $\mathbb{D}$*:*

- $\lambda \mathtt{x}\colon \mathbb{A}\mathtt{.dob\ (boa\ x)} : \mathbb{A} \to \mathbb{D}$
- $\lambda \mathtt{x}\colon \mathbb{A}\mathtt{.doc\ (coa\ x)} : \mathbb{A} \to \mathbb{D}$

To resolve this ambiguity we assume coherence [Pie02] – an additional requirement to the coercion definitions that forces ambiguous transitive coercions to represent the same function. In theorem-provers like Isabelle it is possible to provide a coercion definition system that instructs the user to provide a coherence proof. This procedure is suitable for this purpose.

**Theorem 4.13** (Non-failure)**.** *Coercion insertion will never fail.*

*Intuition.* The algorithm would have failed at one of the previous steps. By construction, $\theta_{\text{sol}}$ is consistent with all arising subtype inequalities. □

**Example 4.14** (example 4.9 continued)**.** *In our example the term* $\mathtt{f}_{[\alpha \to \alpha_3]}\mathtt{True\ 0}$ *was produced during the constraint generation. We have solved the occurring constraints:* $\theta_{sol} = \{\alpha_2 \mapsto \alpha_3, \beta_2 \mapsto \alpha_3 \to \mathbb{B}, \alpha_1 \mapsto \alpha_3, \beta_1 \mapsto \mathbb{B}, \alpha_3 \mapsto \mathbb{N}\}$*. The application of the coercion insertion rules produces the expected result:*

$$\dfrac{\dfrac{\dfrac{\Sigma(\mathtt{f}) = \alpha \to \alpha \to \mathbb{B}}{\Gamma \vdash \mathtt{f}_{[\alpha \to \alpha_3]} \rightsquigarrow \mathtt{f}_{[\alpha \to \mathbb{N}]} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}} \quad \dfrac{\Sigma(\mathtt{True}) = \mathbb{B}}{\Gamma \vdash \mathtt{True} \rightsquigarrow \mathtt{True} : \mathbb{B}} \quad \dfrac{\dfrac{\mathtt{nob} : \mathbb{B} \to \mathbb{N} \in \mathcal{C}}{\mathbb{B} <:_{\mathtt{nob}} \mathbb{N}}}{\vdots}}{\Gamma \vdash \mathtt{f}_{[\alpha \to \alpha_3]}\mathtt{True} \rightsquigarrow \mathtt{f}_{[\alpha \to \mathbb{N}]}(\mathtt{nob}\ \ \mathtt{True}) : \mathbb{N} \to \mathbb{B}} \quad \dfrac{\Sigma(0) = \mathbb{N}}{\Gamma \vdash 0 \rightsquigarrow 0 : \mathbb{N}} \ \mathbb{N} <:_{\mathtt{id}} \mathbb{N}}{\Gamma \vdash \mathtt{f}_{[\alpha \to \alpha_3]}\mathtt{True}\ \ 0 \rightsquigarrow \mathtt{f}_{[\alpha \to \mathbb{N}]}(\mathtt{nob}\ \ \mathtt{True})\,(\mathtt{id}\ \ 0) : \mathbb{B}}$$

# 5. Interaction with type classes

## 5.1. Type classes in Isabelle

The type inference 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 interferes in a more or less subtle way with the type class system of Isabelle. The type class system is described in [Nip93] and [Wen97]. 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 quasiordered (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 $\alpha :: \mathfrak{S}$ for this. A type $T$ can be tested for membership in a sort $\mathfrak{S}$ via the judgement $\vdash_{\text{sort}} T : \mathfrak{S}$. Further, a function $arity(C, \mathfrak{S})$ is given. $arity$ takes a type constructor $C$ of arity $n$ and a sort $\mathfrak{S}$ as input and returns a tuple of $n$ sorts $(\mathfrak{S}_1, \ldots, \mathfrak{S}_n)$. The interpretation of this function is that $\vdash_{\text{sort}} C\ \tau_1 \ldots \tau_n : \mathfrak{S}$ holds only if $\vdash_{\text{sort}} \tau_i : \mathfrak{S}_i$ holds $\forall i = 1 \ldots n$. $arity(C, \mathfrak{S})$ can fail if $C$ constructs types that do not belong to sort $\mathfrak{S}$. Also, a unification that respects the type classes is required. For example, $\alpha :: \mathfrak{S}$ and $\mathbb{X}$ are not unifiable if $\vdash_{\text{sort}} \mathbb{X} : \mathfrak{S}$ does not hold and the unification of $\alpha :: \mathfrak{S}$ and $\beta :: \mathfrak{T}$ is a type variable annotated with the intersection of the sorts $\mathfrak{S}$ and $\mathfrak{T}$.

## 5.2. 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 chapter 4 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 rule 2:

2. Transfer $\alpha :: \mathfrak{S} <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n$ to $C\ (\alpha_1 :: \mathfrak{S}_1)\ (\alpha_2 :: \mathfrak{S}_2)\ \ldots\ (\alpha_n :: \mathfrak{S}_n) <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n$ using fresh variables $\alpha_1 :: \mathfrak{S}_1, \ldots, \alpha_n :: \mathfrak{S}_n$ if $arity(C, \mathfrak{S}) = (\mathfrak{S}_1, \ldots, \mathfrak{S}_n)$ holds. Apply the substitution $\{\alpha :: \mathfrak{S} \mapsto C\ (\alpha_1 :: \mathfrak{S}_1)\ (\alpha_2 :: \mathfrak{S}_2)\ \ldots\ (\alpha_n :: \mathfrak{S}_n)\}$ to the whole constraint set.

## 5.3. Interaction with constraint resolution

Assuming a new unification function that respects the sort annotations the process of building the constraint graph and eliminating cycles will work the same way as before.

The constraint resolution is more problematic. Here, every assignment must be "sort-correct".

**Example 5.1.** *Consider the following two examples in figure 5.1.*

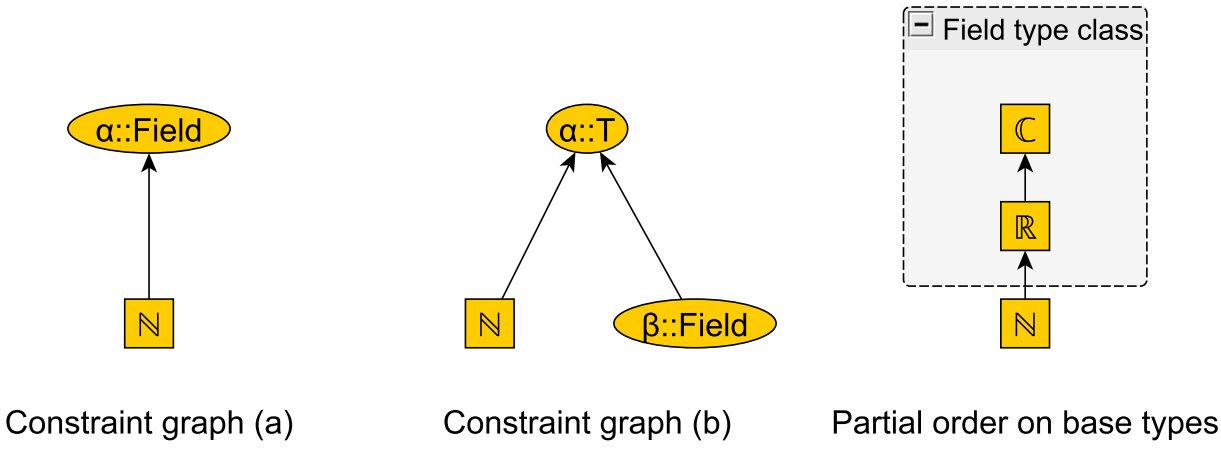



Figure 5.1.: Examples on type class-subtype interference

*In example (a) our algorithm would try to assign the type* $\mathbb{N}$ *to* $\alpha :: Field$*. 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* $\mathbb{R}$*. The second example demonstrates another 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* $\beta :: Field$*. 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 5.2** (Notation)**.** *Let* $S, T$ *be base types,* $X$ *a set of base types,* $\mathfrak{S}$ *a sort and* $\mathfrak{X}$ *a set of sorts. We define*[6] *w.r.t. the given subtype relation "*$<:$*":*

- $\overline{T}^{\mathfrak{X}}_{\mathfrak{S}} = \{T' \mid T <: T',\ T' \text{ base type}, \vdash_{sort} T' : \mathfrak{S},\ \forall \mathfrak{T} \in \mathfrak{X}\ \exists T'' \text{base type}:\ T'' <: T', \vdash_{sort} T'' : \mathfrak{T}\}$ *the set of* ***supertypes*** *that are of sort* $\mathfrak{S}$ *and have subtypes for every sort in* $\mathfrak{X}$
- $\underline{T}^{\mathfrak{X}}_{\mathfrak{S}} = \{T' \mid T' <: T,\ T' \text{ base type}, \vdash_{sort} T' : \mathfrak{S},\ \forall \mathfrak{T} \in \mathfrak{X}\ \exists T'' \text{base type}:\ T' <: T'', \vdash_{sort} T'' : \mathfrak{T}\}$ *the set of* ***subtypes*** *that are of sort* $\mathfrak{S}$ *and have supertypes for every sort in* $\mathfrak{X}$
- ***Supremum*** *of* $S$ *and* $T$*:* $T \sqcup^{\mathfrak{X}}_{\mathfrak{S}} S \in \overline{T}^{\mathfrak{X}}_{\mathfrak{S}} \cap \overline{S}^{\mathfrak{X}}_{\mathfrak{S}}$ *and* $\forall U \in \overline{T}^{\mathfrak{X}}_{\mathfrak{S}} \cap \overline{S}^{\mathfrak{X}}_{\mathfrak{S}} : T \sqcup^{\mathfrak{X}}_{\mathfrak{S}} S <: U$

[6] This definition includes some procedural semantics, i.e. the non-existence of a type that satisfies the supremum/infimum property produces a failure during computation.

- ***Infimum** of $S$ and $T$: $T \sqcap^{\mathfrak{x}}_{\mathfrak{S}} S \in \underline{T}^{\mathfrak{x}}_{\mathfrak{S}} \cap \underline{S}^{\mathfrak{x}}_{\mathfrak{S}}$ and $\forall L \in \underline{T}^{\mathfrak{x}}_{\mathfrak{S}} \cap \underline{S}^{\mathfrak{x}}_{\mathfrak{S}} : L <: T \sqcap^{\mathfrak{x}}_{\mathfrak{S}} S$*
- ***Supremum** of $X$: $\bigsqcup^{\mathfrak{x}}_{\mathfrak{S}} X \in \bigcap_{T \in X} \overline{T}^{\mathfrak{x}}_{\mathfrak{S}}$ and $\forall U \in \bigcap_{T \in X} \overline{T}^{\mathfrak{x}}_{\mathfrak{S}} : \bigsqcup^{\mathfrak{x}}_{\mathfrak{S}} X <: U$*
- ***Infimum** of $X$: $\bigsqcap^{\mathfrak{x}}_{\mathfrak{S}} X \in \bigcap_{T \in X} \underline{T}^{\mathfrak{x}}_{\mathfrak{S}}$ and $\forall L \in \bigcap_{T \in X} \underline{T}^{\mathfrak{x}}_{\mathfrak{S}} : L <: \bigsqcap^{\mathfrak{x}}_{\mathfrak{S}} X$.*

Now, we only need to define how the sort and the set of sorts for the supremum/infimum computation are exactly obtained in the assignment procedure.

CONSTRAINT RESOLUTION ALGORITHM (TYPE CLASSES, A SINGLE ITERATION)

1. For every unassigned variable $\alpha :: \mathfrak{S}$ in $G$ do
   a) Let $P_\alpha$ be the set of all base type predecessors[7] of $\alpha :: \mathfrak{S}$ in $G$
   b) If $P_\alpha = \emptyset$, skip steps c)-f)
   c) Let $\mathfrak{X}_\alpha$ be the set of sorts of all type variable predecessors[7] of $\alpha :: \mathfrak{S}$ in $G$
   d) Let $S_\alpha$ be the set of all base type successors[7] of $\alpha :: \mathfrak{S}$ in $G$
   e) If $\bigsqcup^{\mathfrak{x}_\alpha}_{\mathfrak{S}} P_\alpha$ exists and $\forall$ base types $\beta \in S_\alpha : \bigsqcup^{\mathfrak{x}_\alpha}_{\mathfrak{S}} P_\alpha <: \beta$, update $\theta_{\text{cyc}} \leftarrow \theta_{\text{cyc}} \cup \{\alpha \mapsto \bigsqcup^{\mathfrak{x}_\alpha}_{\mathfrak{S}} P_\alpha\}$ and $G \leftarrow \theta_{\text{cyc}} G$
   f) Fail otherwise.
2. For every unassigned variable $\alpha :: \mathfrak{S}$ in $G$ do
   a) Let $S_\alpha$ be the set of all base type successors[7] of $\alpha :: \mathfrak{S}$ in $G$
   b) If $S_\alpha = \emptyset$, skip steps c)-e)
   c) Let $\mathfrak{X}_\alpha$ be the set of sorts of all type variable successors of $\alpha :: \mathfrak{S}$ in $G$
   d) If $\bigsqcap^{\mathfrak{x}_\alpha}_{\mathfrak{S}} S_\alpha$ exists, update $\theta_{\text{cyc}} \leftarrow \theta_{\text{cyc}} \cup \{\alpha \mapsto \bigsqcap^{\mathfrak{x}_\alpha}_{\mathfrak{S}} P_\alpha\}$ and $G \leftarrow \theta_{\text{cyc}} G$
   e) Fail otherwise.

The properties of the original algorithm are still valid in the type class version. The algorithm still always terminates and is consistent but not complete for arbitrary posets of base types. The completeness is reached by restricting the poset structure. The restriction is even more complex than in the original algorithm because the type class structure is involved. We will discuss the restriction in the next chapter.

[7] This includes also transitive arcs.

# 6. Completeness

## 6.1. Some problematic examples

So far, we have only made statements about termination and consistency of our algorithm. It is equally important to ensure that the algorithm does not fail at a term which can be coerced to a well-typed term. A typing system with this property is called complete.

As already mentioned earlier, our algorithm is not complete for arbitrary posets of base types. This is justified by some simple examples. First, we consider the algorithm for the language without type classes as described in chapter 4.

**Example 6.1.** *The figure 6.1 shows a constraint graph that can not be satisfied with the given base type order using our algorithm.*


Figure 6.1.: Problematic example without type classes

*During constraint resolution the type variable $\alpha$ is the first to be assigned with $\mathbb{R}$ because it is the only variable that has a predecessor. Then, the assignment of $\beta$ will fail, since the infimum $\mathbb{R} \sqcap \mathbb{N}$ does not exist in the given poset. This would not be a problem if the constraints were not satisfiable. Unfortunately, $\{\alpha \mapsto \mathbb{C},\ \beta \mapsto \mathbb{N}\}$ is a solving substitution.*

*Actually, if the algorithm first computes and assigns the infimum of the successors for every variable, this example will work. Of course, this is not a satisfying insight.*

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

**Example 6.2.** *Some problems also occur in the solvable constraint graph in figure 6.2.* $\{\alpha \to \mathbb{Q},\ \beta \to \mathbb{Q}\}$ *is a solving substitution. Again, the presented algorithm shall be applied. In this example it is not clear which variable is assigned first. Our algorithm does not name any specific order for this*[8]*. We assume that* $\beta$ *is the variable to be assigned first.* $\beta$ *gets the value* $\mathbb{N} \sqcup_{\top}^{\{Field\}} \mathbb{Z} = \mathbb{Q}$*. Afterwards, the algorithm tries to assign* $\alpha$ *to* $\bigsqcup_{Field}^{\emptyset}\{\mathbb{N}\}$*. This supremum does not exist, since* $\mathbb{R} \not<: \mathbb{Q}$ *and* $\mathbb{Q} \not<: \mathbb{R}$*. Thus, the algorithm fails.*

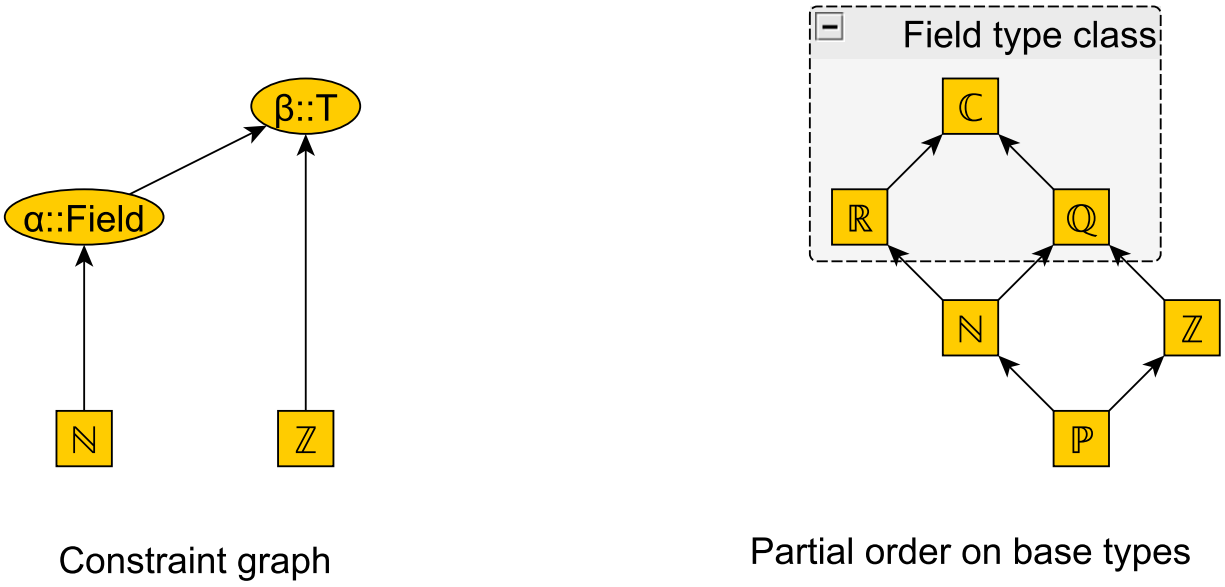



Figure 6.2.: Problematic example with type classes

In both cases, the problem is the non-existence of a supremum/infimum. The solution to this problem is to postulate the existence of the supremum/infimum as a requirement for the partial order of base types. Before we define what this means, we discuss why the problem should be changed and not the algorithm.

## 6.2. Complexity

The question may arise why we are constructing such a sophisticated algorithm if it does not work. The answer is complexity. As the general problem is NP-complete [WO89], we could transform it into a satisfiability problem and let a SAT-solver do the work. However, type inference is a subsystem of a functional programming language that should not be too slow. Transforming the problem into a SAT problem eliminates all the problem-specific information. In our algorithm we use the structure of the subtyping problem splitting it into phases that solve subproblems. Almost all of these subproblems are solved efficiently.

The constraint generation is linear in the length of the term assuming the signature lookup and the generation of fresh variables being constant. The unification is a well known problem that can be solved very efficiently. The most problematic subproblem is the constraint simplification. A cascade of applications of rule 2 may produce an exponential number of new type variables. However, in

[8]The order will actually become unimportant when we introduce the restrictions to solve the completeness problem

practice constraint sets do not behave that bad. Building the graph and solving the constraints is done in polynomial time using graph structures for the constraint set and the partial order on base types.

These complexity considerations are only formulated as conjectures in this thesis. They should be proven formally in a further study.

## 6.3. Necessary restrictions on subtype dependencies

Tiuryn [Tiu92] has shown that satisfiability for an atomic set of subtype constraints can be tested in polynomial time if the partial order on base types is a disjoint union of lattices. The representation of the partial order as a directed graph allows us to speak of weakly connected components. With this terminology Tiuryn's requirement means that every weakly connected component is a lattice. This is exactly the restriction that is needed to ensure that our algorithm from chapter 4 is complete. The completeness is a simple consequence of two facts:

- In a solvable constraint set every supremum/infimum computation does not fail for sets of base types from a single weakly connected component of the restricted base type order.
- Any weakly connected component of a solvable constraint graph may only contain base types from a single weakly connected component of the base type order.

Example 6.2 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 $(\mathcal{P}, <:)$ with any sort $\mathfrak{S}$ to $(\mathcal{P}_{\mathfrak{S}}, <:)$ where $\mathcal{P}_{\mathfrak{S}} = \{\tau \in \mathcal{P} \mid \vdash_{\text{sort}} \tau : \mathfrak{S}\}$ to be a disjoint union of lattices. This restriction extends Tiuryn's restriction, since every type belongs to the sort $\top$ so that it holds $\mathcal{P} = \mathcal{P}_{\top}$.

Unfortunately, this restriction is not strong enough. Example 6.3 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[9] 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 not expect the needed supremum/infimum to exist. In some sense, type class restrictions cut out arbitrary[10] type sets from the assignment candidates. Still, we need to ensure the existence of the supremum/infimum to achieve completeness.

**Example 6.3.** *Assume, the constraint resolution algorithm is applied to the following constraint set:*

$$\{\alpha :: \mathit{All\_but\_rational} <: \mathbb{C},\ \alpha :: \mathit{All\_but\_rational} <: \beta :: \mathit{Rational}\}$$

*Figure 6.3 shows that the assignment candidates do not have a maximal element. So, the infimum does not*

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

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

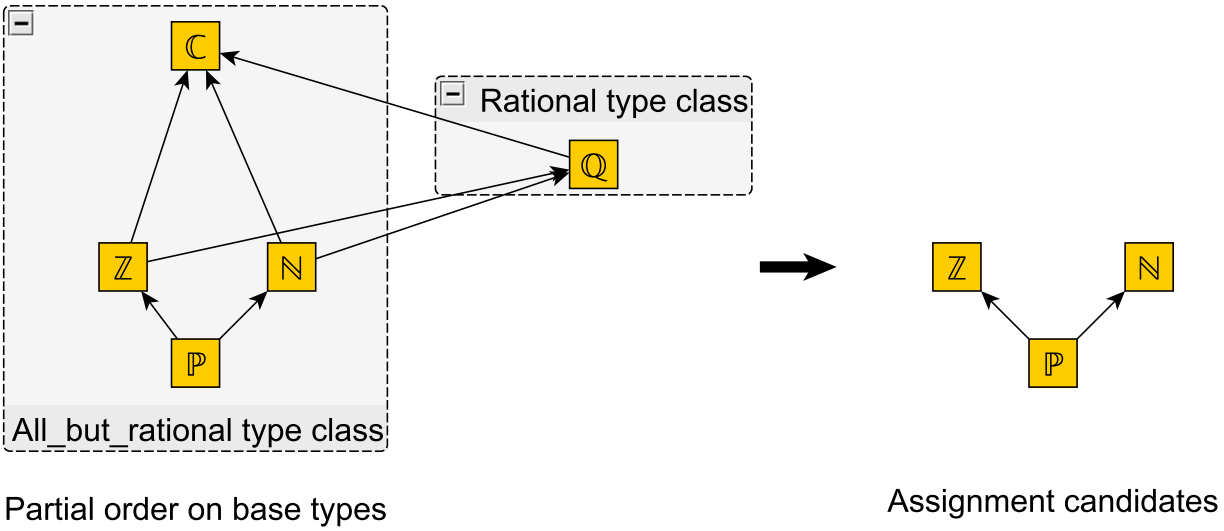



Figure 6.3.: Assignment candidates resulting from computation of $\prod_{All\_but\_rational}^{\{Rational\}}\{\mathbb{C}\}$

*exist and the algorithm fails.*

**Definition 6.4** (Sort-respecting subtype relation)**.** *We call a poset* $(\mathcal{P}, <:)$ ***sort-respecting*** *if* $\forall$ *sets of base types* $X \subseteq \mathcal{P}$*, sets of sorts* $\mathfrak{X}$*, sorts* $\mathfrak{S}$ *:*

- *either* $\bigcap_{T\in X} \underline{T}^{\mathfrak{X}}_{\mathfrak{S}} = \emptyset$ *or* $\bigcap_{T\in X} \underline{T}^{\mathfrak{X}}_{\mathfrak{S}}$ *has a maximal element*[11]
- *and dually either* $\bigcap_{T\in X} \overline{T}^{\mathfrak{X}}_{\mathfrak{S}} = \emptyset$ *or* $\bigcap_{T\in X} \overline{T}^{\mathfrak{X}}_{\mathfrak{S}}$ *has a minimal element*[12]*.*

This definition 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. This is exactly the property that makes our algorithm complete. However, it is not really clear from the definition how strong this restriction is. 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. We can also make the statement that the restriction of the poset to be sort-respecting is weaker than the requirement of linear orders. This statement is justified by the poset in figure 6.4 that is fulfilling our restriction and is obviously not a disjoint union of linear orders.

**Theorem 6.5** (Completeness of constraint resolution)**.** *If the partial order on base types is sort-respecting, constraint resolution will succeed if the input constraint graph is solvable.*

*Informal proof by contradiction.* Assume that the constraint resolution fails for a solvable constraint graph. Constraint resolution fails either during the computation of the supremum/infimum or

[11] i.e. $\exists \tau \in \bigcap_{T\in X} \underline{T}^{\mathfrak{X}}_{\mathfrak{S}} : \forall \sigma \in \bigcap_{T\in X} \underline{T}^{\mathfrak{X}}_{\mathfrak{S}} : \sigma <: \tau$

[12] i.e. $\exists \tau \in \bigcap_{T\in X} \overline{T}^{\mathfrak{X}}_{\mathfrak{S}} : \forall \sigma \in \bigcap_{T\in X} \overline{T}^{\mathfrak{X}}_{\mathfrak{S}} : \tau <: \sigma$

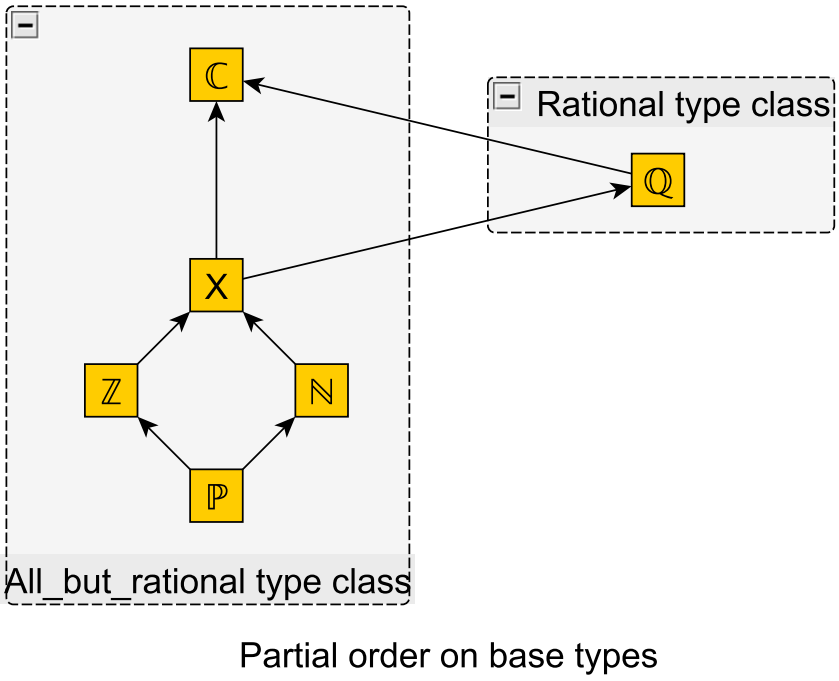



Figure 6.4.: A sort-respecting poset of base types

during the test whether the successors of a variable, which was assigned to the supremum of its predecessors, are supertypes of this supremum.

The first computation can only fail if two types from different weakly connected components of the base type poset are compared, because the supremum/infimum is well-defined according to our restriction of the base type poset in any other case. Clearly, this can never happen in the original solvable graph. Also, this situation is not possible after any assignment step done by our algorithm, since the algorithm will never "leave" the weakly connected component of the base type poset. In other words, if a variable has a neighbour within the weakly connected component $A$ of the base type poset, it will only be assigned to a type from $A$.

So the failure must occur in the successor test in step 1d of the constraint resolution. Let $\alpha :: \mathfrak{S}$ be the variable that produces the failure. We use the notation $P_\alpha$, $S_\alpha$ and $\mathfrak{X}_\alpha$ as defined in the algorithm. Thus, a base type $\beta \in S_\alpha$ with the property $\beta <: \bigsqcup_{\mathfrak{S}}^{\mathfrak{X}_\alpha} P_\alpha$, $\beta \neq \bigsqcup_{\mathfrak{S}}^{\mathfrak{X}_\alpha} P_\alpha$ exists. Since $\bigsqcup_{\mathfrak{S}}^{\mathfrak{X}_\alpha} P_\alpha$ is the smallest upper bound for $P_\alpha$, $\beta$ can not be an upper bound for $P_\alpha$ w.r.t. the sorts $\mathfrak{S}$ and $\mathfrak{X}_\alpha$. Since the original constraint graph does not contain such contradictions, a previous assignment of any base type $\tau$ within $P_\alpha \cup \{\beta\}$ must have been wrong. However, all of these previous assignments tested whether $\tau <: \beta$ and $\forall \sigma \in P_\alpha : \ \sigma <: \tau$ and passed these tests. This contradicts the assumption. □

**Theorem 6.6** (Completeness)**.** *If the partial order on base types is sort-respecting, the presented algorithm will always coerce the term whenever it is possible.*

*Intuition.* Using theorem 6.5 and theorem 4.13 we only have to show that the constraint generation and preprocessing are producing a solvable constraint graph if the input term can be coerced. This rather technical proof by contradiction assuming a failure for a term that can be coerced and the

subsequent case distinction of every algorithm step where the failure could have appeared is left to the reader. □

# 7. Conclusion

## 7.1. Usage in Isabelle

As the presented algorithm is used to extend the type inference system of the Isabelle proof assistant, some interesting aspects of the concrete implementation are mentioned below.

### 7.1.1. User interface for subtyping

We provide two commands to define a coercion and a map function for a type constructor. Both commands take a single term as their only argument.

The map functions are stored in a table. Using the type of a map function, the variance vector[13] can be computed if the given function is a valid map function. This vector is also stored in the map function table.

The partial order on base types is represented as a graph in our implementation. The command to add coercions only checks whether the graph stays acyclic. More interesting restrictions like coherence and lattice order are not controlled yet. An approach that would suit Isabelle well is to formulate these restrictions as a proof goal and let the user provide the proof.

### 7.1.2. Ambiguous coercions

Isabelle uses rewriting rules to transform and simplify proof goals during the process of interactive proving. These rules are very sensitive when it comes to syntax. A misplaced coercion would disrupt the proof process. Even though we assured that our algorithm only produces type-correct terms, a coercion could occur at an unexpected[14] place.

Consider the following example.

**Example 7.1.** *We inspect the term* `sin(1 + 1)`[15] *with the given type signatures* $\Sigma(\texttt{sin}) = \mathbb{R} \to \mathbb{R}$, $\Sigma(+) = \alpha :: Number \to \alpha :: Number \to \alpha :: Number$ *and* $\Sigma(\texttt{1}) = \mathbb{N}$. *Further* $\mathbb{N} <:_{\texttt{ron}} \mathbb{R}$ *holds.*

*There are two possibilities to repair the ill-typed term.*

1. `sin(ron (1 + 1))`

2. `sin((ron 1) + (ron 1))`

[13] A vector that denotes the variance for every argument of a type constructor
[14] At least unexpected for the Isabelle simplifier that applies the rewriting rules
[15] + is used as an infix operator but should be regarded as a usual binary function

*Our algorithm will insert coercions the way it is done in the first possibility. If we invert the order of handling the predecessors/successors in the constraint resolution, the algorithm will produce the second term. The proceeding of the algorithm in this example can be found in appendix B.*

Of course our algorithm is deterministic. It will always insert coercions at the same place for a single term. However, if the exact insertion point is ambiguous, the explicit type conversion is still the preferable way in this case. Otherwise, some rewriting rules possibly can not be applied, although semantically, it should be possible.

### 7.1.3. Benchmarks

A type inference algorithm is useless if it is too slow. We compare the original Isabelle type inference algorithm with our implementation of coercive subtyping. For this purpose, we consider the Isabelle theory `src/HOL/Decision_Procs/Approximation.thy` from the Isabelle repository[16]. This theory contains statements about bounds of numerical approximations of real-valued functions. In these statements and their proofs a lot of type conversions, e.g. integer to real numbers, are required. With the new type inference algorithm, it is possible to omit most of these conversions, defining them as coercions.

To inspect the run-time behaviour of our algorithm we consider four different settings.

1. Original Isabelle type inference, unchanged `Approximation.thy`.
2. New algorithm, unchanged `Approximation.thy`.
3. New algorithm, added coercion and map function definitions in `Approximation.thy`.
4. New algorithm, added coercion and map function definitions in `Approximation.thy`, removed most type conversions from the source code.

All four settings were tested on the same hardware (2,0 GHz Intel®Core™2 Duo CPU with 2 GB RAM running Ubuntu 10.04). The CPU usage time for every setting is charted in table 7.1.

| setting | CPU usage in $min : sec$ |
|---|---|
| 1 | $4 : 34.85$ |
| 2 | $5 : 02.31$ |
| 3 | $5 : 06.63$ |
| 4 | $5 : 01.27$ |

Table 7.1.: Run-time benchmarks

`Approximation.thy` is one of the larger and more complex theories in the Isabelle/HOL repository. Still, our new subtyping algorithm does not need a displeasingly long time to step through this theory.

[16] `http://isabelle.in.tum.de/repos/isabelle/`

## 7.2. Further extensions

The implementation is intended to be the foundation of subtyping in Isabelle. The algorithm can serve as an entry point for theoretical and practical research in subtype polymorphism. In this final section two possible fields of further studies are presented.

### 7.2.1. Arbitrary subtype orderings

We have seen that the presented algorithm only works under certain restrictions. An interesting question is how strong these restrictions are. For example, an intuitive way to model subtype relations between numeric types is a linear order, which is even a stronger restriction than we require. A study of type families that behave well in terms of subtype orderings as, e.g. the numeric types, is desirable. Further it is possible to analyze what can be done if the problematic partial orders cannot be avoided. At this point the problem probably becomes NP-complete. That should also be investigated and proven.

### 7.2.2. Subtype relation for types with different type constructors

Another thinkable research field is the further extension of the subtype relation. An interesting approach is to allow coercions between constructed types with different type constructor. A concrete example is a possible coercion between the list and the function type constructors. Any list with elements of type $\alpha$ could be declared as a subtype of the function type $\mathbb{N} \to \alpha$ using the function `nth`[17] of type $List\ \alpha \to \mathbb{N} \to \alpha$. The programmer could then write `xs 42` instead of `nth xs 42` for an arbitrary list `xs`.

It is not immediately clear how the presented algorithm has to be modified in order to use this extension. Also, the usability has to be considered. Type inference supports the user in writing correct programs. A type error is often the first indicator that the source code contains bugs. With the proposed extension a lot more terms become type correct. Still, it is essential that a user receives an error message in case of a bug rather than the source code gets transformed into something nobody ever intended.

[17] `nth xs 42` returns the 42th element of the list `xs`

# Bibliography


[BM96] François Bourdoncle and Stephan Merz. 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, Paris, March 1996.

[dt06] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2006. Version 8.1.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *ESOP*, pages 94–114, 1988.

[KL03] Robert Kießling and Zhaohui Luo. Coercions in hindley-milner systems. In *TYPES*, pages 259–275, 2003.

[LM92] Patrick Lincoln and John C. Mitchell. Algorithmic aspects of type inference with subtypes. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–304, New York, NY, USA, 1992. ACM.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[Mit84] John C. Mitchell. Coercion and type inference. In *POPL*, pages 175–185, 1984.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *J. Funct. Program.*, 1(3):245–285, 1991.

[Nip93] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. CUP, 1993.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[Tiu92] Jerzy Tiuryn. Subtype inequalities. In *LICS*, pages 308–315, 1992.

[Wen97] Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.

[WO89] Mitchell Wand and Patrick O'Keefe. On the complexity of type inference with coercion. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 293–298, New York, NY, USA, 1989. ACM.

# A. Constraint pipeline

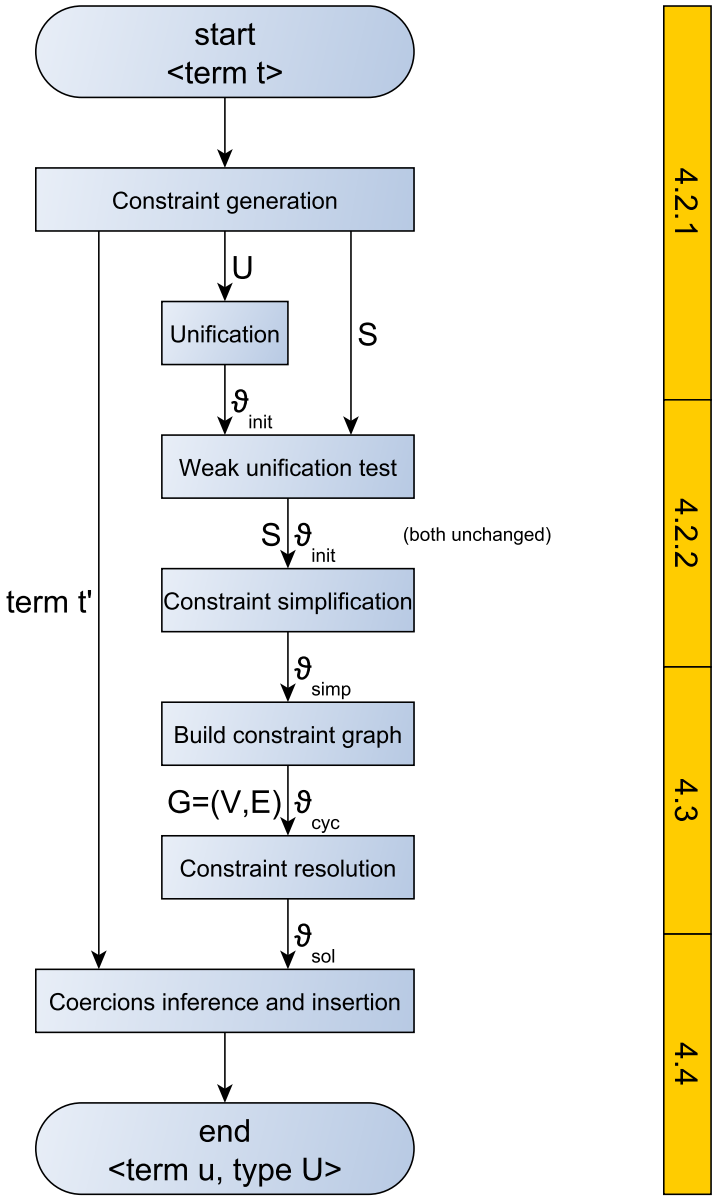



Figure A.1.: Abstract control flow of the subtyping algorithm

# B. Inference of the sinus example

We infer the type of the term `sin(1 + 1)` with signatures and coercions described in chapter 7. The notation $\alpha^*$ is used as an abbreviation for $\alpha :: Number$.

$$\dfrac{\dfrac{\dfrac{\Sigma(+_{[\alpha\mapsto\alpha_4]}) = \alpha^* \to \alpha^* \to \alpha^*}{\Gamma \vdash +_{[\alpha\mapsto\alpha_4]} : \alpha_4^* \to \alpha_4^* \to \alpha_4^* \nabla(\emptyset,\emptyset)} \quad \dfrac{\Sigma(1) = \mathbb{N}}{\Gamma \vdash 1 : \mathbb{N}\nabla(\emptyset,\emptyset)} \quad \overbrace{\alpha_3, \beta_3}^{\text{fresh}}}{\Gamma \vdash 1 \ +_{[\alpha\mapsto\alpha_4]} : \beta_3 \nabla \left(\{\alpha_4^* \to \alpha_4^* \to \alpha_4^* \doteq \alpha_3 \to \beta_3\}, \{\mathbb{N} <: \alpha_3\}\right)} \quad \vdots \quad \dfrac{\Sigma(1) = \mathbb{N}}{\Gamma \vdash 1 : \mathbb{N}\nabla(\emptyset,\emptyset)} \quad \overbrace{\alpha_2, \beta_2}^{\text{fresh}}}{\Gamma \vdash 1 \ +_{[\alpha\mapsto\alpha_4]} 1 : \beta_2 \nabla \left( \left\{ \begin{array}{c} \alpha_4^* \to \alpha_4^* \to \alpha_4^* \doteq \alpha_3 \to \beta_3, \\ \beta_3 \doteq \alpha_2 \to \beta_2 \end{array} \right\}, \left\{ \begin{array}{c} \mathbb{N} <: \alpha_3, \\ \mathbb{N} <: \alpha_2 \end{array} \right\} \right)} \quad \vdots \quad \dfrac{\Sigma(\texttt{sin}) = \mathbb{R} \to \mathbb{R}}{\Gamma \vdash \texttt{sin} : \mathbb{R} \to \mathbb{R} \nabla(\emptyset,\emptyset)} \quad \overbrace{\alpha_1, \beta_1}^{\text{fresh}}$$

$$\Gamma \vdash \texttt{sin(1} \ +_{[\alpha\mapsto\alpha_4]} \texttt{1)} : \beta_1 \nabla \left( \left\{ \begin{array}{c} \alpha_4^* \to \alpha_4^* \to \alpha_4^* \doteq \alpha_3 \to \beta_3, \\ \beta_3 \doteq \alpha_2 \to \beta_2, \\ \mathbb{R} \to \mathbb{R} \doteq \alpha_1 \to \beta_1 \end{array} \right\}, \left\{ \begin{array}{c} \mathbb{N} <: \alpha_3, \\ \mathbb{N} <: \alpha_2, \\ \beta_2 <: \alpha_1 \end{array} \right\} \right)$$

$\theta_{\text{init}} = \{\alpha_3 \mapsto \alpha_4^*,\ \beta_3 \mapsto \alpha_4^* \to \alpha_4^*,\ \alpha_2 \mapsto \alpha_4^*,\ \beta_2 \mapsto \alpha_4^*,\ \alpha_1 \mapsto \mathbb{R},\ \beta_1 \mapsto \mathbb{R}\}$ is a most general unifier of the unification constraints $\{\alpha_4^* \to \alpha_4^* \to \alpha_4^* \doteq \alpha_3 \to \beta_3,\ \beta_3 \doteq \alpha_2 \to \beta_2,\ \mathbb{R} \to \mathbb{R} \doteq \alpha_1 \to \beta_1\}$. Applying this substitution to the subtype constraints produces an already atomic constraint set with the corresponding constraint graph as shown in figure B.1.

Our algorithm will first compute the supremum of the predecessors of $\alpha_4 :: Number$. This results in the assignment $\alpha_4 :: Number \mapsto \mathbb{N}$. However, if the same algorithm first covers the successors, the other valid assignment will be produced: $\alpha_4 :: Number \mapsto \mathbb{R}$. Of course, different assignments result in different terms as it is visible below.

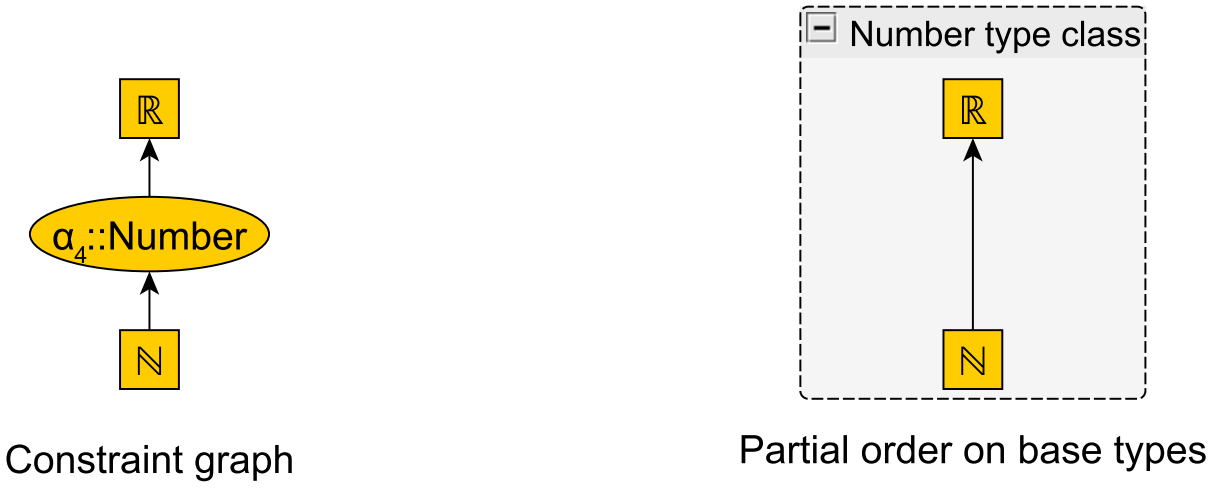



Figure B.1.: Ambiguous constraint graph

In case of the substitution $\alpha_4 :: Number \mapsto \mathbb{N}$ coercion insertion proceeds as following:

$$\dfrac{\dfrac{\dfrac{\Sigma(+_{[\alpha\mapsto\alpha_4]}) = \alpha^* \to \alpha^* \to \alpha^*}{\Gamma \vdash +_{[\alpha\mapsto\alpha_4]} \rightsquigarrow +_{[\alpha\mapsto\mathbb{N}]} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}} \quad \dfrac{\Sigma(1) = \mathbb{N}}{\Gamma \vdash 1 \rightsquigarrow 1 : \mathbb{N}} \quad \mathbb{N} <:_{\mathtt{id}} \mathbb{N}}{\Gamma \vdash 1\ +_{[\alpha\mapsto\alpha_4]} \rightsquigarrow (\mathtt{id}\ 1)\ +_{[\alpha\mapsto\mathbb{N}]} : \mathbb{N} \to \mathbb{N}} \quad \dfrac{\Sigma(1) = \mathbb{N}}{\Gamma \vdash 1 \rightsquigarrow 1 : \mathbb{N}} \quad \mathbb{N} <:_{\mathtt{id}} \mathbb{N}}{\Gamma \vdash 1\ +_{[\alpha\mapsto\alpha_4]} 1 \rightsquigarrow (\mathtt{id}\ 1)\ +_{[\alpha\mapsto\mathbb{N}]} (\mathtt{id}\ 1) : \mathbb{N}} \quad \mathbb{N} <:_{\mathtt{ron}} \mathbb{R}}{\Gamma \vdash \mathtt{sin}(1\ +_{[\alpha\mapsto\alpha_4]} 1) \rightsquigarrow \mathtt{sin}(\mathtt{ron}\ ((\mathtt{id}\ 1)\ +_{[\alpha\mapsto\mathbb{N}]} (\mathtt{id}\ 1))) : \mathbb{R}}$$

Substitution $\alpha_4 :: Number \mapsto \mathbb{R}$ results in an other derivation tree:

$$\dfrac{\dfrac{\dfrac{\Sigma(+_{[\alpha\mapsto\alpha_4]}) = \alpha^* \to \alpha^* \to \alpha^*}{\Gamma \vdash +_{[\alpha\mapsto\alpha_4]} \rightsquigarrow +_{[\alpha\mapsto\mathbb{R}]} : \mathbb{R} \to \mathbb{R} \to \mathbb{R}} \quad \dfrac{\Sigma(1) = \mathbb{N}}{\Gamma \vdash 1 \rightsquigarrow 1 : \mathbb{N}} \quad \mathbb{N} <:_{\mathtt{ron}} \mathbb{R}}{\Gamma \vdash 1\ +_{[\alpha\mapsto\alpha_4]} \rightsquigarrow (\mathtt{ron}\ 1)\ +_{[\alpha\mapsto\mathbb{R}]} : \mathbb{R} \to \mathbb{R}} \quad \dfrac{\Sigma(1) = \mathbb{N}}{\Gamma \vdash 1 \rightsquigarrow 1 : \mathbb{N}} \quad \mathbb{N} <:_{\mathtt{ron}} \mathbb{R}}{\Gamma \vdash 1\ +_{[\alpha\mapsto\alpha_4]} 1 \rightsquigarrow (\mathtt{ron}\ 1)\ +_{[\alpha\mapsto\mathbb{R}]} (\mathtt{ron}\ 1) : \mathbb{R}} \quad \mathbb{R} <:_{\mathtt{id}} \mathbb{R}}{\Gamma \vdash \mathtt{sin}(1\ +_{[\alpha\mapsto\alpha_4]} 1) \rightsquigarrow \mathtt{sin}(\mathtt{id}\ ((\mathtt{ron}\ 1)\ +_{[\alpha\mapsto\mathbb{R}]} (\mathtt{ron}\ 1))) : \mathbb{R}}$$