Extending Hindley-Milner Type Inference with Coercive Structural Subtyping

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 according to the standard Hindley-Milner system (without any subtypes). The algorithm is sound and, if the subtype relation on base types is a disjoint union of lattices, also complete. 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 §4), 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 standard 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. In §4, we formulate the correctness and completeness statements and discuss restrictions on the subtype relation that are necessary to prove them. An outline of related research is given in §5.

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 α, β, ...), 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* Σ 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 τ is a *subtype* of σ. The subtyping relation that we consider in this paper is *structural*: if $\tau <: \sigma$, then τ and σ 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 τ to be used in a context where a term of the supertype σ 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 Γ and a term t, returns a new term u augmented with coercions, together with a type τ, 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 [12, §16.2]. 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 θ, which is a function mapping type variables to types such that $\theta\alpha \neq \alpha$ for only finitely many α. 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 θ and a type τ for a given term t and context Γ 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 θ 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. Then, it is easy to see that 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.

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, §17.12]. 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 4.

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 5. Given a term t and a context Γ, the algorithm returns a type τ, 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.

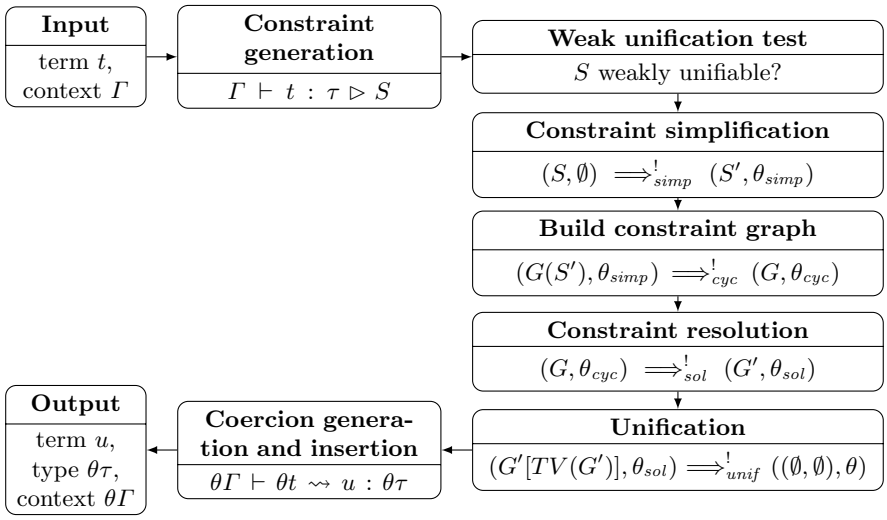

Fig. 4. Top-level control flow of the subtyping algorithm

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. 5. 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 βs 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 6. 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*

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

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

$$var^i_C(\tau, \sigma) = \tau \doteq \sigma.$$

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.

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 (α, β 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', θ_{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 inequation $T <: U$ for base types T and U such that T is not a subtype of U or an equation $\tau \doteq \sigma$ such that τ and σ are not unifiable. In the latter three 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

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

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

EXPAND-L
$(\{\alpha <: C\ \tau_1\ \ldots\ \tau_n\} \uplus S, \theta) \Longrightarrow_{simp} (\theta'\ (\{\alpha <: C\ \tau_1\ \ldots\ \tau_n\} \cup S), \theta' \circ \theta)$
where $\theta' = \{\alpha \mapsto C\ \alpha_1\ \ldots\ \alpha_n\}$
and $\alpha_1\ \ldots\ \alpha_n$ are fresh variables

EXPAND-R
$(\{C\ \tau_1\ \ldots\ \tau_n <: \alpha\} \uplus S, \theta) \Longrightarrow_{simp} (\theta'\ (\{C\ \tau_1\ \ldots\ \tau_n <: \alpha\} \cup S), \theta' \circ \theta)$
where $\theta' = \{\alpha \mapsto C\ \alpha_1\ \ldots\ \alpha_n\}$
and $\alpha_1\ \ldots\ \alpha_n$ are fresh variables

ELIMINATE
$(\{U <: T\} \uplus S, \theta) \Longrightarrow_{simp} (S, \theta)$
where U, T are base types
and $U <: T$

Fig. 6. Rule-based constraint simplification $\Longrightarrow_{simp}$

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. Weak unification differs from standard unification in that it identifies base types, which is necessary since two types τ and σ 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 θ 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*

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

and T_0 is a fixed base type not used elsewhere.

Weak unification is merely used as a termination-test in our algorithm before constraint simplification (see Figure 4).

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 (σ, τ). 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 $\sigma \preceq: \tau$ for the subtyping relation on base types induced by the set of coercions $\mathcal{C}$, which is defined by

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

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 7.

Cycle-Elim

$$((V, E), \theta) \quad \Longrightarrow_{cyc} \quad ((V \setminus K \cup \{\tau_K\}, E' \cup P \times \{\tau_K\} \cup \{\tau_K\} \times S), \theta_K \circ \theta)$$

where K is a cycle in (V, E)
and $\theta_K = mgu(K)$
and $\{\tau_K\} = \theta_K K$
and $E' = \{(\tau, \sigma) \in E \mid \tau \notin K, \sigma \notin K\}$
and $P = \{\tau \mid \exists \sigma \in K.\ (\tau, \sigma) \in E\} \setminus K$
and $S = \{\sigma \mid \exists \tau \in K.\ (\tau, \sigma) \in E\} \setminus K$

Fig. 7. Rule-based cycle elimination $\Longrightarrow_{cyc}$

Figure 8 visualizes an example of cycle elimination. We call the substitution obtained from cycle elimination θ_{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 [20]. First, we define some basic lattice-theoretic notions.

Definition 7. *Let S, T, T' denote base types and X a set of base types. With respect to the given subtype relation $\preceq:$ we define:*

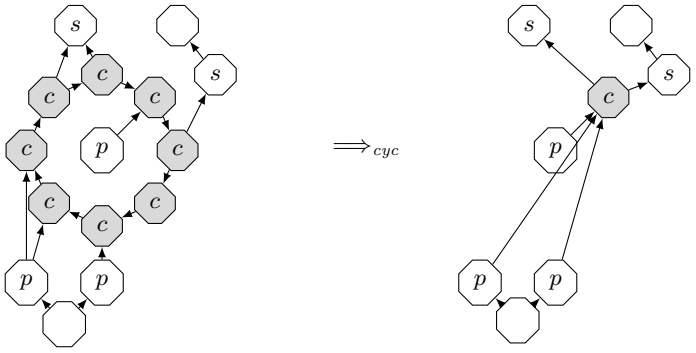

Fig. 8. Collapse of a cycle in a graph

- $\overline{T} = \{T' \mid T \preceq: T'\}$, *the set of* supertypes
- $\underline{T} = \{T' \mid T' \preceq: T\}$, *the set of* subtypes
- $T \sqcup S \in \overline{T} \cap \overline{S}$ *and* $\forall U \in \overline{T} \cap \overline{S}.\ T \sqcup S \preceq: U$, *the* supremum *of* S *and* T
- $T \sqcap S \in \underline{T} \cap \underline{S}$ *and* $\forall L \in \underline{T} \cap \underline{S}.\ L \preceq: T \sqcap S$, *the* infimum *of* S *and* T
- $\bigsqcup X \in \bigcap_{T \in X} \overline{T}$ *and* $\forall U \in \bigcap_{T \in X} \overline{T}.\ \bigsqcup X \preceq: U$, *the* supremum *of* X
- $\bigsqcap X \in \bigcap_{T \in X} \underline{T}$ *and* $\forall L \in \bigcap_{T \in X} \underline{T}.\ L \preceq: \bigsqcap X$, *the* infimum *of* X.

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

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

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

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 9. The resulting substitution is referred to as θ_{sol}.

The original algorithm described by Wand and O'Keefe [20] 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^G_α is empty. It assigns each type variable α the infimum $\bigsqcap S^G_\alpha$ of its upper bounds, and then checks whether the assigned type is greater than all lower bounds P^G_α. If $\bigsqcap S^G_\alpha$ does not exist, their algorithm fails. If S^G_α 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. As we do not want to annotate the term with unresolved subtype constraints, all variables in a single weakly connected component should be unified. This is done by the rule Unify-WCC shown in Figure 10 and produces the final substitution θ.

$$\textsc{Assign-Sup}\quad (G,\theta) \Longrightarrow_{sol} \big(\{\alpha \mapsto \bigsqcup P_\alpha^G\}G, \{\alpha \mapsto \bigsqcup P_\alpha^G\} \circ \theta\big) \text{ if } \alpha \in TV(G) \wedge P_\alpha^G \neq \emptyset \wedge \exists \bigsqcup P_\alpha^G \wedge \forall T \in S_\alpha^G.\ \bigsqcup P_\alpha^G \preceq: T$$

$$\textsc{Fail-Sup}\quad (G,\theta) \Longrightarrow_{sol} \text{FAIL} \text{ if } \alpha \in TV(G) \wedge P_\alpha^G \neq \emptyset \wedge (\nexists \bigsqcup P_\alpha^G \vee \exists T \in S_\alpha^G.\ \bigsqcup P_\alpha^G \not\preceq: T)$$

$$\textsc{Assign-Inf}\quad (G,\theta) \Longrightarrow_{sol} \big(\{\alpha \mapsto \bigsqcap S_\alpha^G\}G, \{\alpha \mapsto \bigsqcap S_\alpha^G\} \circ \theta\big) \text{ if } \alpha \in TV(G) \wedge S_\alpha^G \neq \emptyset \wedge \exists \bigsqcap S_\alpha^G \wedge \forall T \in P_\alpha^G.\ T \preceq: \bigsqcap S_\alpha^G$$

$$\textsc{Fail-Inf}\quad (G,\theta) \Longrightarrow_{sol} \text{FAIL} \text{ if } \alpha \in TV(G) \wedge S_\alpha^G \neq \emptyset \wedge (\nexists \bigsqcap S_\alpha^G \vee \exists T \in P_\alpha^G.\ T \not\preceq: \bigsqcap S_\alpha^G)$$

Fig. 9. Rule-based constraint resolution $\Longrightarrow_{sol}$

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

Fig. 10. Rule-based WCC-unification $\Longrightarrow_{unif}$

Example 2. Going back to Example 1, we apply our algorithm to the term $leq_{[\alpha \mapsto \alpha_3]}\ n\ i$. According to the inference rules from Figure 5, we obtain $\Gamma \vdash leq_{[\alpha \mapsto \alpha_3]}\ n\ i : \beta_1 \rhd \{\alpha_3 \to \alpha_3 \to \mathbb{B} \doteq \alpha_2 \to \beta_2, \beta_2 \doteq \alpha_1 \to \beta_1, \mathbb{N} <: \alpha_2, \mathbb{Z} <: \alpha_1\}$. Simplifying the generated 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 $\{\mathbb{N} <: \alpha_3, \mathbb{Z} <: \alpha_3\}$. This yields the constraint graph shown in Figure 11. The constraint resolution algorithm assigns α_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 θ, the final result, is θ_{sol}.

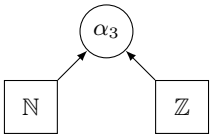

Fig. 11. Constraint graph of $leq_{[\alpha \mapsto \alpha_3]}\ n\ i$

In §4 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 θ. 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 §4.

4 Total correctness and completeness

To prove total correctness, we need to show that for any input t and Γ, the algorithm either returns a substitution θ, 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, 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. Due to space limitations, all proofs and supporting lemmas had to be omitted and can be found in the full version of this paper [19]. Since the reduction rules in each phase are applied nondeterministically, the algorithm may output different substitutions for the same input term t and context Γ. By $\mathit{AlgSol}(\Gamma, t)$ we denote the set of all such substitutions.

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

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

Theorem 2 (Termination). *The algorithm terminates for any input t and Γ.*

4.1 An example 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.

Example 3. Figure 12 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 α is assigned first, it will receive value $\mathbb{R}$. Then, the assignment of β 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 β is assigned before α is practically irrelevant because we cannot possibly exhaust all nondeterministic choices.

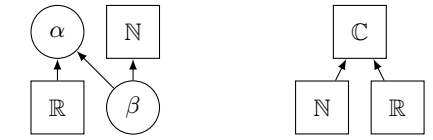

Fig. 12. Problematic example without type classes

We see that the problem is the non-existence of a supremum or infimum. The solution to this problem 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.

4.2 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.

Therefore, we assume in the following that the base type poset is a disjoint union of lattices.

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

Definition 8 (Equality modulo coercions). *Two substitutions θ and θ' 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 9 (Subsumed). *The substitution θ' is* subsumed modulo coercions *wrt. to the type variable set X by the substitution θ, if there exists a substitution δ such that $\theta' \approx_X \delta \circ \theta$. We write $\theta \lesssim_X \theta'$.*

Let $TV(\tau)$ and $TV(t)$ be the sets of type variables that occur in τ 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 3 (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* $\theta \lesssim_{TV(\Gamma) \cup TV(t)} \theta'$*.*

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

Example 4. 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 α_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.

5 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 [20]. 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. Pottier [13] describes a sound but incomplete

simplification procedure for subtype constraints. 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 [11].

5.1 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. The subtype lattice in this theory is a linear order of the 3 types *nat*, *int*, *real*.

Isabelle supports an extension of Hindley-Milner polymorphism with type classes [10]. In the full version of this paper [19], we cover type classes, too, and show how to extend our algorithms soundly; completeness seems difficult to achieve in this context.

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 $\{\alpha \preceq: \mathbb{R},\ \alpha \preceq: \beta,\ \mathbb{N} \preceq: \beta\}$. 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 $\mathbb{R} \to \mathbb{R}$. A simple-minded extension of our algorithm to *let* might choose one of the two substitutions to type u and would necessarily fail. 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, there is no complete typing

algorithm for *let* terms that simply inserts coercions. You may need to expand or otherwise transform *let* first.

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. Nordlander, J.: Polymorphic subtyping in O'Haskell. Sci. Comput. Program. 43(2-3), 93–127 (2002)
12. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)
13. Pottier, F.: Simplifying subtyping constraints: a theory. Information & Computation 170(2), 153–183 (Nov 2001)
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. Traytel, D., Berghofer, S., Nipkow, T.: Extending Hindley-Milner type inference with coercive subtyping (long version) (2011), `www.in.tum.de/~nipkow/pubs/coercions.pdf`
20. 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)