ELSEVIER Science of Computer Programming 39 (2001) 57–92 Programming
www.elsevier.nl/locate/scico

Finite subtype inference with explicit polymorphism

Dominic Duggan

Department of Computer Science, Stevens Institute of Technology, Castle Point on the Hudson, Hoboken, NJ 07030, USA

Abstract

Finite subtype inference occupies a middle ground between Hindley–Milner-type inference (as in ML) and subtype inference with recursively constrained types. It refers to subtype inference where only finite types are allowed as solutions. This approach avoids some open problems with general subtype inference, and has practical motivation where recursively constrained types are not appropriate. This paper presents algorithms for finite subtype inference, including checking for entailment of inferred types against explicitly declared polymorphic types. This resolves for finite types a problem that is still open for recursively constrained types. Some motivation for this work, particularly for finite types and explicit polymorphism, is in providing subtype inference for first-class container objects with polymorphic methods. © 2001 Elsevier Science B.V. All rights reserved.

1. Introduction

Type inference is the process of statically type-checking a program where some or all of the type information has been omitted from the program text. ML and Haskell are examples of programming languages where type inference has been a spectacular success. The particular flavor of type inference used by ML and Haskell is *Hindley–Milner-type inference* [17]. The type-checker accumulates equality constraints via a tree walk of the abstract syntax tree, and then uses a unification algorithm to compute a (most general) unifying substitution for these constraints.

More recently, attention has been focused on *subtype inference* [2,3,7,20,23,29]. With this work, the type-checker accumulates subtype constraints while traversing the abstract syntax tree, and then applies a constraint solver to check these constraints for consistency. Pottier [23] and Smith and Trifonov [29] have considered the problem of entailment in these type systems, which is important for example in interface matching. Subtype inference continues to be an important avenue of research, particularly in simplifying inferred types to make them practically useful.

E-mail address: dduggan@cs.stevens-tech.edu (D. Duggan).

PII: S0167-6423(00)00013-7

Hindley–Milner-type inference and subtype inference represent two extremes in the type inference continuum:

Hindley–Milner	Finite subtype	Subtype
Equality	Subtyping	Subtyping
Finite types	Finite types	Infinite types
Inferred monotypes	Inferred monotypes	Inferred monotypes
Inferred polytypes	{ Inferred polytypes / Specified polytypes* }	Inferred polytypes

Between these two extremes, there is an intermediate point: *finite subtype inference*. While this alternative allows subtyping and type subsumption, it does not assume that types are potentially infinite trees (as with the most recent work on subtype inference).

Why should we consider subtype inference with finite types? It is worth recalling why ML, for example, does not allow circular types (types as potentially infinite trees). The problem was pointed out by Solomon [28]: the problem of deciding the equality of parameterized recursive types is equivalent to the problem of deciding the equality of deterministic context-free languages (DCFLs). Although this problem has recently been shown to be decidable [25], it is not clear if an efficient algorithm for checking this equivalence is possible. This problem is avoided in ML-type inference by making the folding and unfolding of recursive types explicit (using data constructors and pattern matching, respectively), so that circular types are not needed.

A motivation for infinite types in subtype inference is to support objects with recursive interfaces. However, the problem discovered by Solomon also holds for recursive interfaces for container objects. Consider, for example, a `set` object with interface:

$$\begin{aligned}\texttt{set}(\alpha) = \{&\texttt{map} : \forall\beta.(\alpha \to \beta) \to \texttt{set}(\beta),\\ &\texttt{product} : \forall\beta.\texttt{set}(\beta) \to \texttt{set}(\alpha * \beta),\\ &\texttt{power} : \forall\beta.\texttt{unit} \to \texttt{set}(\texttt{set}(\alpha))\}\end{aligned}$$

All of the methods in this interface are examples of non-regular recursion in the object interface. In another paper [5], we consider an object design for languages with type inference that avoids this problem. The approach there is to again make the folding and unfolding of recursive object interfaces explicit, in object creation and method invocation, respectively.

Structural subtyping refers to subtype systems where the only subtype relation is between atomic types, and types in the subtype relation have a common tree structure. Early work on subtype inference considered structural subtyping with finite types [9,15,18]. However there are non-trivial differences between finite subtype inference with structural subtyping and with record containment. For example even checking for finiteness is markedly different, as elaborated upon in Section 5.

Another design point is whether polymorphic types should be inferred or specified. All of the work so far assumes that polymorphic types are inferred. The disadvantage of these approaches is that the inferred types are large and complex, diminishing their

practical usefulness, despite recent work on simplifying inferred types [7,23,29]. One way to avoid this problem is to require that the programmer provide explicit interfaces for polymorphic functions. This approach introduces fresh technical complications of its own. In Hindley–Milner-type inference, *mixed-prefix unification* has been used to control scoping of type variables with explicit polymorphic-type declarations (an idea originally used by Leroy and Mauny [14], and subsequently by Odersky and Läufer [19]). In this paper we extend subtype inference with *constraint-solving under a mixed prefix*, in order to support subtype inference with explicit polymorphism.

Explicit polymorphism also derives motivation from our work on container objects with recursive interfaces [5]. We avoid the problems with first-class polymorphism in Hindley–Milner-type inference, by requiring explicit-type specifications on polymorphic methods. This is similar to the use of universal types to incorporate impredicativity into Hindley–Milner-type inference [12,19,24], but tied to the object system instead of to datatypes (again because we are concerned with container objects with polymorphic methods).

Even if we are not concerned with polymorphic methods, explicit polymorphism may be required if we wish to provide a type-checking algorithm for some of the object-type systems considered by Abadi and Cardelli [1]. For example, Abadi and Cardelli [1, Section 16.2] consider a type system with primitive covariant self types, incorporating a type rule for method update that requires that the new method definition be parametric in the type of self (the type of self is a type parameter constrained by the object interface).

Explicit polymorphism requires that it be possible to check for entailment of inferred types from declared types. For infinite types this is problematic. Although incomplete algorithms have been published [23,29], the decidability of entailment remains open [29]. Heinglein and Rehof [10,11] prove that there are entailment problems that may be as hard for finite trees as for infinite trees in the presence of non-structural order with $\bot$ and $\top$; this happens intuitively because finite trees ordered non-structurally can be used to approximate infinite trees arbitrarily. In this paper we demonstrate that entailment is decidable for a particular variant of finite subtyping, giving further motivation for our approach.

Our type system incorporates non-structural subtyping, because it includes record containment. However, we deliberately omit the $\top$ and $\bot$ types, the supertype of all types and the subtype of all types, respectively. This is in contrast with the aforesaid work [23,29]. Our motivation for this is that we obtain a fine-grained-type system that flags-type errors consistently with Hindley–Milner-type inference. Matters change markedly if we add $\top$ to the type system, consider for example:

```
if true then (λx.x) else 3
```

With the $\top$ type, there is no static-type error here. On the other hand, the absence of $\top$ and $\bot$ does lead to some complications in the constraint-checking algorithms. Our algorithms are not immediately applicable to a type system with $\top$ and $\bot$. For example the constraints $\{\alpha \leqslant (\beta * \beta), \alpha \leqslant \beta\}$ are finitely satisfiable (instantiate α to $\bot$).

Palsberg et al. [21] consider the problem of constraint-solving (checking for consistency and finiteness) for a type system with $\top$ and $\bot$ and structural subtyping, while Palsberg [20] considers how these algorithms can be generalized to non-structural subtyping. The following table summarizes the relationship between this work and that of others (further comparison is provided in Section 9):

	Mitchell [18]	ST [29]	Pottier [23]	PWO [21]	Sequeiera [26]	This Paper
Finite subtyping	√			√	√	√
$\top, \bot$ types		√	√	√		
Universal types					√	√
Non-struct subtyping		√	√	√		√

Section 2 introduces our type system. We do not overburden the paper with any details of the object system mentioned earlier [5], but present a familiar ML-like language with record-based subtyping and explicit polymorphism. Section 4 provides the type inference algorithm. Section 4 provides algorithms for checking consistency and entailment; these algorithms must be defined mutually recursively. Section 5 considers the check for finite solutions; perhaps surprisingly, this check must be done after consistency and entailment checking. Section 6 considers the use of mixed prefix constraint-solving to check that there are well-scoped solutions. Section 7 considers the soundness and completeness of type inference.

Section 8 considers a semantic notion of entailment between constrained polymorphic types. The applicability of this approach is in interface matching in module systems. This section demonstrates that our algorithms are complete decision procedures for deducing entailment for subtype constraints over finite types. Finally, Section 9 considers further related work and provides conclusions.

2. Type system

The mini-language we consider is a language with functions, pairs and records. Subtyping is based on containment between record types. This is extended contravariantly to function types and covariantly to product types. Polymorphic types allow quantified type variables to be constrained by upper bounds. We use $\forall \overline{\alpha_n} \leqslant \overline{\tau_n}.\tau$ generically for a sequence of quantifiers where all, none or some of the variables may have upper bounds.

$$\tau \in \text{Simple Type} ::= \alpha \mid \texttt{int} \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$
$$\sigma \in \text{Poly Type} ::= \forall \alpha.\sigma \mid \forall \alpha \leqslant \tau.\sigma \mid \tau$$
$$e \in \text{Exp} ::= x \mid \lambda x.e \mid (e_1\ e_2) \mid (e_1, e_2) \mid \textbf{fst}\ e \mid \textbf{snd}\ e \mid$$
$$\texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2 \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l$$

As noted in the previous section, our type system does not have a $\top$ or a $\bot$ type.

$$\frac{A(x) = \forall\overline{\alpha_n} \leq \overline{\tau_n}.\tau \quad \Gamma \vdash \tau'_i \leq \{\overline{\tau'_{i-1}/\alpha_{i-1}}\}\tau_i \text{ for } i = 1,\ldots,n}{\Gamma; A \vdash x : \{\overline{\tau'_n/\alpha_n}\}\tau} \quad \text{(VAR)}$$

$$\frac{\Gamma; A \vdash e_1 : (\tau_2 \to \tau_1) \quad \Gamma; A \vdash e_2 : \tau_2}{\Gamma; A \vdash (e_1\ e_2) : \tau_1} \quad \text{(APP)}$$

$$\frac{\Gamma; A, x : \tau_1 \vdash e : \tau_2}{\Gamma; A \vdash (\lambda x.e) : (\tau_1 \to \tau_2)} \quad \text{(ABS)}$$

$$\frac{\Gamma; A \vdash e_1 : \tau_1 \quad \Gamma; A \vdash e_2 : \tau_2}{\Gamma; A \vdash (e_1, e_2) : (\tau_1 * \tau_2)} \quad \text{(PAIR)}$$

$$\frac{\Gamma; A \vdash e : (\tau_1 * \tau_2)}{\Gamma; A \vdash \mathbf{fst}\ e : \tau_1} \quad \text{(FST)}$$

$$\frac{\Gamma; A \vdash e : (\tau_1 * \tau_2)}{\Gamma; A \vdash \mathbf{snd}\ e : \tau_2} \quad \text{(SND)}$$

$$\frac{\Gamma; A \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma; A \vdash e_n : \tau_n}{\Gamma; A \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \quad \text{(RECORD)}$$

$$\frac{\Gamma; A \vdash e : \{l : \tau\}}{\Gamma; A \vdash e.l : \tau} \quad \text{(SELECT)}$$

$$\frac{\begin{array}{c}\Gamma, \overline{\alpha_n} \leq \overline{\tau_n}; A \vdash e_1 : \tau_1 \quad \Gamma; A, x : (\forall\overline{\alpha_n} \leq \overline{\tau_n}.\tau_1) \vdash e_2 : \tau_2 \\ FV(\tau_i) \cap \{\alpha_i, \ldots, \alpha_n\} = \{\} \text{ for } i = 1,\ldots,n\end{array}}{\Gamma; A \vdash (\texttt{let}\ x : (\forall\overline{\alpha_n} \leq \overline{\tau_n}.\tau_1) = e_1\ \texttt{in}\ e_2) : \tau_2} \quad \text{(LET)}$$

$$\frac{\Gamma; A \vdash e : \tau \quad \Gamma \vdash \tau \leq \tau'}{\Gamma; A \vdash e : \tau'} \quad \text{(SUB)}$$

Fig. 1. Type rules.

The type rules are specified in Fig. 1 using judgements of the form $\Gamma; A \vdash e : \tau$. The context A is a sequence of program variable bindings $(x : \sigma)$, while Γ is a sequence of type variable bindings $((\alpha \leqslant \tau)$, or just α where there is no upper bound specified in a type annotation). The VAR rule includes the case where $n = 0$ and the type of x is a simple type τ. The type rules include a type subsumption rule SUB; the subtype rules are provided in Fig. 2.

The main construct of interest is the `let` construct. This allows generalization of types to polymorphic types, but requires an explicit-type annotation. This is demonstrated by the LET-type rule in Fig. 1. It is also possible to define a monomorphic version of the let, that does not require a type annotation:

$$(\texttt{letmono}\ x = e_1\ \texttt{in}\ e_2) \equiv (\lambda x.e_2)\ e_1.$$

A type (tree) can be considered as a prefix-closed partial function from *paths* π to types (subterms). Paths are sequences from the alphabet $\{d, r, f, s\} \cup \mathit{FieldName}$. d

$$\frac{\Gamma = \Gamma_1, \alpha \leq \tau, \Gamma_2}{\Gamma \vdash \alpha \leq \tau} \quad \text{(SUBAX)}$$

$$\Gamma \vdash \tau \leq \tau \quad \text{(SUBREFL)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2 \quad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \quad \text{(SUBTRANS)}$$

$$\frac{\Gamma \vdash \tau_1' \leq \tau_1 \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash (\tau_1 \to \tau_2) \leq (\tau_1' \to \tau_2')} \quad \text{(SUBFUN)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_1' \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash (\tau_1 * \tau_2) \leq (\tau_1' * \tau_2')} \quad \text{(SUBPAIR)}$$

$$\frac{\Gamma \vdash \tau_i \leq \tau_i' \text{ for } i = 1, \ldots, n \text{ where } m \geq n}{\Gamma \vdash \{\overline{l_m : \tau_m}\} \leq \{\overline{l_n : \tau_n'}\}} \quad \text{(SUBREC)}$$

Fig. 2. Subtype rules.

and r denote the domain and range, respectively, of a function type, while f and s denote the first and second component types, respectively, of a product type. $dom(\tau) = \{\pi | \tau(\pi) \text{ is defined}\}$. The *subtree* at $\pi \in dom(\tau)$ is the function $\pi' \mapsto \tau(\pi\pi')$. A tree τ is finite if $dom(\tau)$ is finite, and infinite otherwise.

An *assignment* θ on $V \subseteq TyVar$ is a total function mapping-type variables in V to types. It is homomorphically extended to simple types τ with $FV(\tau) \subseteq V$: $\theta(\texttt{int}) = \texttt{int}$, $\theta(\tau_1 \to \tau_2) = \theta(\tau_1) \to \theta(\tau_2)$, $\theta(\tau_1 * \tau_2) = \theta(\tau_1) * \theta(\tau_2)$, $\theta(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) = \{l_1 : \theta(\tau_1), \ldots, l_n : \theta(\tau_n)\}$. We denote composition of assignments by juxtaposition: $\theta'\theta(\tau) = \theta'(\theta(\tau))$.

3. Type inference

The type inference algorithm is provided in Fig. 3. It uses judgements of the form $\mathcal{Q}; A \vdash e : \tau \textit{ with } \mathcal{Q}', C$. $\mathcal{Q}$ and $\mathcal{Q}'$ are *quantifier prefixes*, while C is a constraint set, described by

$$\mathcal{Q} ::= \varepsilon \mid \exists\alpha \mid \forall\alpha \mid \forall\alpha \leqslant \tau \mid \mathcal{Q}_1.\mathcal{Q}_2$$
$$C ::= \{\} \mid \{\tau_1 \leqslant \tau_2\} \mid C_1 \cup C_2$$

The quantifier prefix records "flexible" (existential) and "rigid" (universal) variables introduced during type inference, while the relative ordering records the scope of the type variables (existential α can be instantiated to a type containing universal β only if β is quantified to the left of α in the quantifier prefix). To be well formed, the concatenation of quantifier prefixes $\mathcal{Q}_1.\mathcal{Q}_2$ requires that the variables bound in the quantifier prefixes $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are distinct. Define $EV(\mathcal{Q}) = \{\alpha | (\exists\alpha) \in \mathcal{Q}\}$ and $UV(\mathcal{Q}) = \{\alpha | (\forall\alpha) \in \mathcal{Q}$

$$\frac{A(x) = \forall\overline{\alpha_m} \leq \overline{\tau_m}.\tau \quad Q' = Q.\exists\overline{\alpha_m}}{Q;A \vdash x : \tau \textbf{ with } Q',\ \{\alpha_i \leq \tau_i \mid i = 1,\ldots,m\}} \quad \text{(VAR)}$$

$$\frac{Q.\exists\alpha;A,x : \alpha \vdash e : \tau \textbf{ with } Q',\ C}{Q;A \vdash (\lambda x.e) : (\alpha \to \tau) \textbf{ with } Q',\ C} \quad \text{(ABS)}$$

$$\frac{Q;A \vdash e_1 : \tau_1 \textbf{ with } Q_1,\ C_1 \quad Q_1;A \vdash e_2 : \tau_2 \textbf{ with } Q_2,\ C_2}{Q;A \vdash (e_1\ e_2) : \alpha \textbf{ with } (Q_2.\exists\alpha),\ C_1 \cup C_2 \cup \{\tau_1 \leq (\tau_2 \to \alpha)\}} \quad \text{(APP)}$$

$$\frac{Q;A \vdash e_1 : \tau_1 \textbf{ with } Q_1,\ C_1 \quad Q_1;A \vdash e_2 : \tau_2 \textbf{ with } Q_2,\ C_2}{Q;A \vdash (e_1, e_2) : (\tau_1 * \tau_2) \textbf{ with } Q_2,\ C_1 \cup C_2} \quad \text{(PAIR)}$$

$$\frac{Q;A \vdash e : \tau \textbf{ with } Q',\ C}{Q;A \vdash (\textbf{fst}\ e) : \alpha \textbf{ with } (Q'.\exists\alpha.\exists\beta),\ C \cup \{\tau \leq (\alpha * \beta)\}} \quad \text{(FST)}$$

$$\frac{Q;A \vdash e : \tau \textbf{ with } Q',\ C}{Q;A \vdash (\textbf{snd}\ e) : \beta \textbf{ with } (Q'.\exists\alpha.\exists\beta),\ C \cup \{\tau \leq (\alpha * \beta)\}} \quad \text{(SND)}$$

$$\frac{Q_0 = Q \quad Q_{i-1};A \vdash e_i : \tau_i \textbf{ with } Q_i,\ C_i \text{ for } i = 1,\ldots,n}{Q;A \vdash \{l_1 = e_1,\ldots,l_n = e_n\} : \{l_1 : \tau_1,\ldots,l_n : \tau_n\} \textbf{ with } Q_n,\ C_1 \cup \cdots \cup C_n} \quad \text{(RECORD)}$$

$$\frac{Q;A \vdash e : \tau \textbf{ with } Q',\ C}{Q;A \vdash e.l : \alpha \textbf{ with } (Q'.\exists\alpha),\ C \cup \{\tau \leq \{l : \alpha\}\}} \quad \text{(SELECT)}$$

$$\frac{\begin{array}{c}(Q.\forall\overline{\alpha_n} \leq \overline{\tau_n});A \vdash e_1 : \tau_1 \textbf{ with } Q_1,\ C_1 \\ Q_1;A,x : (\forall\overline{\alpha_n} \leq \overline{\tau_n}.\tau) \vdash e_2 : \tau_2 \textbf{ with } Q_2,\ C_2\end{array}}{Q;A \vdash (\texttt{let}\ \ x : (\forall\overline{\alpha_n} \leq \overline{\tau_n}.\tau) = e_1\ \ \texttt{in}\ \ e_2) : \tau_2 \textbf{ with } Q_2,\ C_1 \cup C_2 \cup \{\tau_1 \leq \tau\}} \quad \text{(LET)}$$

Fig. 3. Type inference.

or $(\forall\alpha \leqslant \tau) \in \mathscr{Q}\}$, then the condition is

$$\mathscr{Q}_1.\mathscr{Q}_2 \text{ is well-formed if } (EV(\mathscr{Q}_1) \cup UV(\mathscr{Q}_1)) \cap (EV(\mathscr{Q}_2) \cup UV(\mathscr{Q}_2)) = \{\}.$$

The inputs to the algorithm are $\mathscr{Q}$, A and e. The outputs are the inferred type τ, a set of constraints C constraining instantiations of A and τ, and an extension $\mathscr{Q}'$ of the quantifier prefix $\mathscr{Q}$. While type constraints C are percolated upward from the leaves to the root of an execution of the type inference algorithm, the quantifier prefix $\mathscr{Q}$ is threaded through the execution of the algorithm in a depth-first manner (and therefore fixes the order of execution of the algorithm). One reason for this is to ensure that any new variables introduced during type inference are unique, another reason is to maintain the relative order of existential and universal variables (which is important for scoping of type variables).

We will need to reason about satisfying substitutions for constraints under a mixed prefix. We use the following technical device (introduced by Duggan [6], and extending the unification logic originally introduced by Miller [16]):

Definition 1 (*Constraint logic with substitutions*). A term $\mathscr{F}$ of the *constraint logic* is a pair $\mathscr{Q}C$ where $\mathscr{Q}$ is a quantifier prefix, and where C is a set of constraints.

Derivability for the judgement form $\theta \models \Gamma \vdash \mathscr{F}$ is defined by the following rules:

$$\frac{\Gamma \vdash C}{\theta \models \Gamma \vdash \varepsilon C}$$

$$\frac{\alpha \notin dom(\theta) \cup dom(\Gamma) \quad \theta \models \Gamma, \alpha \leqslant \tau \vdash \mathscr{F}}{\theta \models \Gamma \vdash (\forall \alpha \leqslant \tau . \mathscr{F})}$$

$$\frac{\alpha \in dom(\theta) \quad FV(\theta(\alpha)) \subseteq dom(\Gamma) \quad \Gamma \vdash_{\Sigma} \theta\alpha : \chi \quad \theta \models \Gamma \vdash \{\theta(\alpha)/\alpha\}\mathscr{F}}{\theta \models \Gamma \vdash (\exists \alpha . \mathscr{F})}$$

Unification logic was originally introduced by Miller [16] in order to reason about the correctness of unification under a mixed prefix. Miller used a complicated mechanism based on decomposing a substitution into the composition of a sequence of single-variable substitutions. Our construction takes a simpler approach, extending the unification logic to a type system for reasoning about the correctness of satisfying substitutions. In particular, the third rule requires that, in the substitution for an existental variable α, this substitution only contains as free variables those variables that are universally quantified to the left of α.

Definition 2 (*Satisfiability*)**.** Given a quantifier prefix $\mathscr{Q}$ and constraint formula C. θ *satisfies* $\mathscr{Q}$ *and* C if $\theta \models \{\} \vdash \mathscr{Q}C$. Denote this by $\theta \models \mathscr{Q}C$. $\mathscr{Q}$ and C are *satisfiable* if there is some substitution θ such that $\theta \models \mathscr{Q}C$. Denote that $\mathscr{Q}$ and C are satisfiable by $\models \mathscr{Q}C$.

The following notion will be useful:

$$UVARS\text{-}CONTEXT(\mathscr{Q}) = \{\alpha \,|\, (\forall \alpha) \in \mathscr{Q}\} \cup \{(\alpha \leqslant \tau) \,|\, (\forall \alpha \leqslant \tau) \in \mathscr{Q}\}.$$

Definition 3 (*Ground satisfiability*)**.** A constraint formula C is *ground satisfiable* if there is some substitution θ mapping the free variables (both universal and existential) of C to ground types, such that $\vdash \tau \leqslant \tau'$ for all $(\tau \leqslant \tau') \in \theta(C)$.

Lemma 4. *If* $\mathscr{Q}$ *and* C *are satisfiable, then they are ground satisfiable.*

The converse is not necessarily true. Consider for example $\exists \alpha . \forall \beta \ (\alpha \leqslant \beta)$, which is ground satisfiable (with the substitution $\{\tau/\alpha, \tau/\beta\}$ for any ground type τ), but is not satisfiable. Our notion of satisfiability captures important scoping information that is missing from other notions based on ground satisfiability.

4. Containment and entailment

The first step in constraint-checking is to verify that there is a satisfying substitution for the accumulated constraints, ignoring finiteness and well scoping in the satisfying

$$(C,C_0) \Longrightarrow_Q (C \cup \{(\tau_1 \le \tau_2)\}) \text{ if } C_0 = \{(\tau_1 \le \alpha),(\alpha \le \tau_2)\} \quad (1)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{(\tau_1' \le \tau_1),(\tau_2 \le \tau_2')\}) \text{ if } C_0 = \{((\tau_1 \to \tau_2) \le (\tau_1' \to \tau_2'))\} \quad (2)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{(\tau_1 \le \tau_1'),(\tau_2 \le \tau_2')\}) \text{ if } C_0 = \{((\tau_1 * \tau_2) \le (\tau_1' * \tau_2'))\} \quad (3)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{(\tau_i \le \tau_i') \mid i = 1,\ldots,n\}) \text{ if } C_0 = \{(\{\overline{l_m} : \overline{\tau_m}\} \le \{\overline{l_n} : \overline{\tau_n'}\})\} \text{ and } m \ge n \quad (4)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau \le \tau'\}) \text{ if } Q = Q_1 \forall \alpha \le \tau Q_2 \text{ and } C_0 = \{(\alpha \le \tau')\},\ \tau' \notin EV(Q) \quad (5)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\beta \sqcap \tau\}) \text{ if } C_0 = \{(\tau \le \beta)\},\ \beta \in EV(Q) \quad (6)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\beta \sqcup \tau\}) \text{ if } C_0 = \{(\beta \le \tau)\},\ \beta \in EV(Q) \quad (7)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau_1' \sqcup \tau_1, \tau_2 \sqcap \tau_2'\}) \text{ if } C_0 = \{((\tau_1 \to \tau_2) \sqcap (\tau_1' \to \tau_2'))\} \quad (8)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau_1 \sqcap \tau_1', \tau_2 \sqcap \tau_2'\}) \text{ if } C_0 = \{((\tau_1 * \tau_2) \sqcap (\tau_1' * \tau_2'))\} \quad (9)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau_1' \sqcap \tau_1, \tau_2 \sqcup \tau_2'\}) \text{ if } C_0 = \{((\tau_1 \to \tau_2) \sqcup (\tau_1' \to \tau_2'))\} \quad (10)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau_1 \sqcup \tau_1', \tau_2 \sqcup \tau_2'\} C_0) \text{ if } C_0 = \{((\tau_1 * \tau_2) \sqcup (\tau_1' * \tau_2'))\} \quad (11)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{(\tau_i \sqcup \tau_j') \mid l_i = l_j'\}) \text{ if } C_0 = \{(\{\overline{l_m} : \overline{\tau_m}\} \sqcup \{\overline{l_n} : \overline{\tau_n'}\})\} \quad (12)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau \sqcap \tau'\}) \text{ if } \alpha \in EV(Q) \text{ and } C_0 = \{(\alpha \sqcap \tau),(\alpha \sqcap \tau')\} \quad (13)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau \sqcup \tau'\}) \text{ if } \alpha \in EV(Q) \text{ and } C_0 = \{(\alpha \sqcup \tau),(\alpha \sqcup \tau')\} \quad (14)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau \sqcap \tau'\}) \text{ if } Q = Q_1 \forall \alpha \le \tau Q_2 \text{ and } C_0 = \{(\alpha \sqcap \tau')\},\ \tau' \notin TyVar \quad (15)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau \le \tau'\}) \text{ if } Q = Q_1 \forall \alpha \le \tau Q_2 \text{ and } C_0 = \{(\alpha \sqcup \tau')\},\ \tau' \notin TyVar \quad (16)$$

$$(C,C_0) \Longrightarrow_Q (C \cup \{\tau \sqcap \tau'\}) \text{ if } C_0 = \{(\alpha \sqcap \beta)\},\ (\forall \alpha \le \tau),(\forall \beta \le \tau') \in Q, \text{ and } Q \not\forall \alpha \textbf{ alias } \beta \quad (17)$$

Fig. 4. Entailment algorithm.

substitution. We need to check that the constraints are consistent, and are entailed by the hypothetical bounds on the universally quantified variables.

Definition 5 (*Entailment algorithm*)**.** Given $\mathcal{Q}$ and C, the containment and entailment algorithm is provided in Fig. 4 using transitions of the form $C \Rightarrow_{\mathcal{Q}} (C', C_0)$. A step in the algorithm is described by the following rule:

$$\frac{C_0 \subseteq C \quad C_0 \notin L \quad (C,C_0) \Rightarrow_{\mathcal{Q}} C'}{(C,L) \Rightarrow_{\mathcal{Q}} (C', L \cup \{C_0\})}$$

Define $(C,L) \stackrel{*}{\Rightarrow}_{\mathcal{Q}} (C',L')$ to denote the repeated application of the algorithm to (C,L) until it reaches a fixed point (C',L'). Let $SUBTYPE\text{-}CLOS(\mathcal{Q}C)$ denote C' where (C',L) is the result of applying the algorithm to $(C,\{\})$.

Rules (1)–(4) are the usual subtype closure, combining transitive closure with downward closure conditions, and check for containment in the original constraint set. The

latter conditions must be satisfied for the original subtype constraints to be satisfied. The remaining transition checks for entailment of the accumulated constraints from the hypotheses. Rule (5) checks that an upper bound on a universal variable can be satisfied:

$$(C, C_0) \Rightarrow_{\mathcal{Q}} (C \cup \{\tau \leqslant \tau'\}) \quad \text{if } \mathcal{Q} = \mathcal{Q}_1.\forall\alpha \leqslant \tau.\mathcal{Q}_2$$

and

$$(\alpha \leqslant \tau') \in C, \quad \tau' \notin EV(\mathcal{Q}).$$

Intuitively the constraint $(\alpha \leqslant \tau')$, where α is universal and $\tau' \neq \alpha$ is not an existential variable, can only be deduced through applications of SubAx and SubTrans, where the "cut" formula is the upper bound τ for α. This rule adds the constraint $\tau \leqslant \tau'$, the second premise in a derivation ending with an application of SubTrans.

There is a duality between universal (rigid) variables and existential (flexible) variables. For a constrained existential variable $\alpha \leqslant \tau$, constraint checking needs to ensure that there is *some* instantiation for α that satifies this and the other constraints. For a constrained universal variable $\alpha \leqslant \tau$, constraint checking needs to ensure that this constraint is satisfied by *any* instantiation for α that is a subtype of the upper bound τ_α of α. This latter condition is satisfied if τ_α is a subtype of τ.

Rules (5) and (16) are noteworthy for the fact that they generate new subtype constraints, and this is the reason that the containment and entailment algorithms must be defined mutually recursively. Consider, for example,

$$\forall\alpha \leqslant (\{x : \mathtt{int}\} * \{x : \mathtt{int}\}), \exists\beta.\exists\gamma\{\alpha \leqslant (\beta * \gamma),\ \beta \leqslant \{x : \mathtt{int}\},\ \gamma \leqslant \{y : \mathtt{int}\}\}.$$

Entailment generates the constraint

$$(\{x : \mathtt{int}\} * \{x : \mathtt{int}\}) \leqslant (\beta * \gamma)$$

and containment generates the constraints

$$\{x : \mathtt{int}\} \leqslant \beta,\ \{x : \mathtt{int}\} \leqslant \gamma,\ \{x : \mathtt{int}\} \leqslant \{x : \mathtt{int}\},\ \{x : \mathtt{int}\} \leqslant \{y : \mathtt{int}\}.$$

and the last of these constraints violates containment.

Rules (6) and (7) check for compatibility of upper and lower bounds on existential variables. "Compatibility" means that we do not have incompatible non-variable bounds on variables, such as $((\tau_1 * \tau_2) \leqslant \beta), ((\tau_1' \to \tau_2') \leqslant \beta) \in C$,

$$(C, C_0) \Rightarrow_{\mathcal{Q}} (C \cup \{\beta \sqcap \tau\}) \quad (\tau \leqslant \beta) \in C,\ \beta \in EV(\mathcal{Q}) \qquad (6)$$

$$(C, C_0) \Rightarrow_{\mathcal{Q}} (C \cup \{\beta \sqcup \tau\}) \quad (\beta \leqslant \tau) \in C,\ \beta \in EV(\mathcal{Q}) \qquad (7)$$

The predicates $\tau \sqcap \tau'$ and $\tau \sqcup \tau'$ are used to check for meet compatibility and join compatibility, respectively. They are generated by rules (6) and (7).

Rules (13) and (14) perform transitive closure on the join and meet compatibility constraints. We implicitly assume that the join and meet compatibility predicates are

commutative, so the algorithm treats constraints as being equivalent up to commutativity of $\sqcup$ and $\sqcap$.

Rule (13) performs transitive closure on meet compatibility constraints. For example, given the constraints

$$\tau \leqslant \alpha, \quad \alpha \leqslant \beta, \quad \tau' \leqslant \beta.$$

From this we generate the constraints

$$\tau \leqslant \beta, \quad \tau \sqcap \beta, \quad \tau' \sqcap \beta, \quad \tau \sqcap \tau'.$$

As another example, consider

$$\alpha \leqslant (\mathtt{int} * \beta), \quad \alpha \leqslant (\mathtt{int} * \{l : \tau\}), \quad \gamma \leqslant (\mathtt{int} * \beta), \quad \gamma \leqslant (\mathtt{int} * \{l : \tau'\}).$$

Applying rules (6)–(14) generates the constraints

$$\beta \sqcup \{l : \tau\}, \quad \beta \sqcup \{l : \tau'\}, \quad \tau \sqcup \tau'.$$

As an example of a limitation of these checks, if we consider

$$\alpha \leqslant (\mathtt{int} * \beta), \quad \alpha \leqslant (\mathtt{int} * \tau), \quad (\mathtt{int} * \beta) \leqslant \gamma, \ (\mathtt{int} * \tau') \leqslant \gamma.$$

Then the algorithm generates the constraints

$$(\mathtt{int} * \beta) \sqcup (\mathtt{int} * \tau), \quad (\mathtt{int} * \beta) \sqcap (\mathtt{int} * \tau'), \quad \beta \sqcup \tau, \ \beta \sqcap \tau',$$

these checks do not ensure that the types τ and τ' with which β must be join and meet compatible are in turn compatible. This check is performed by the match check in the next section. The reason for the join and meet checks, as part of the containment check in this section, is to ensure that record types that must be join compatible have compatible shared fields, and that a variable bounded above by universal variables has a least upper bound.

Rules (8)–(12) distribute $\sqcap$ and $\sqcup$ over the type constructors. There is no rule for analyzing $\{\overline{l_m} : \overline{\tau_m}\} \sqcap \{\overline{l_n} : \overline{\tau'_n}\}$, analogous to Rule (12) for $\{\overline{l_m} : \overline{\tau_m}\} \sqcup \{\overline{l_n} : \overline{\tau'_n}\}$. Two record types that have an existential variable as a common upper bound are always compatible, since they have the common upper bound $\{\ \}$.

We also need to check for compatibility of universal variables with other types in lower and upper bounds, for example $((\alpha * \tau_1) \leqslant \beta), \ ((\tau' * \tau_2) \leqslant \beta) \in C$, where α is universal (with upper bound τ) and β existential. Rule (15) is used to check meet compatibility of universal variables with non-variable types:

$$(C, C_0) \Rightarrow_{\mathscr{Q}} (C \cup \{\tau \sqcap \tau'\}) \quad \text{if } \mathscr{Q} = \mathscr{Q}_1 \forall \alpha \leqslant \tau \mathscr{Q}_2$$

and

$$(\alpha \sqcap \tau') \in C, \quad \tau' \notin \mathit{TyVar}$$

For example, given

$$\mathcal{Q} = \mathcal{Q}_1.\forall\alpha\leqslant\tau.\mathcal{Q}_2 \quad \text{and} \quad C \supseteq \{\alpha\leqslant\beta, \tau'\leqslant\beta\},$$

where β is existential and τ' non-variable, compatibility of the latter two constraints requires that $(\tau\leqslant\beta)$ and $(\tau'\leqslant\beta)$ be compatible. The fact that β has a non-variable lower bound τ' means that any satisfying substitution must instantiate it to a supertype of τ'. Since α, a universal variable, must be a subtype of this supertype (because of the constraint $\alpha\leqslant\beta$), it must be the case that the upper bound τ of α is a subtype of this supertype. This in turn requires that τ and τ' be meet compatible.

Rule (16) is used to check join compatibility of universal variables with non-existential-variable types:

$$(C, C_0) \Rightarrow_{\mathcal{Q}} (C \cup \{\tau\leqslant\tau'\}) \quad \text{if } \mathcal{Q} = \mathcal{Q}_1\forall\alpha\leqslant\tau\mathcal{Q}_2$$

and

$$(\alpha \sqcup \tau') \in C, \quad \tau' \notin \mathit{TyVar}.$$

For example, compatibility of $(\beta\leqslant\alpha)$ and $(\beta\leqslant\tau')$ requires that $\tau\leqslant\tau'$. In this case, the constraint $\beta\leqslant\alpha$ requires that β be instantiated to some variable lower bound of α in any satisfying substitution, since α is a universal variable with only variable lower bounds. Then the constraint $\beta\leqslant\tau'$ can only be satisfied if the upper bound τ of α is a subtype of τ'.

Rule (17) checks for meet compatibility of two universal variables:

$$(C, C_0) \Rightarrow_{\mathcal{Q}} (C \cup \{\tau \sqcap \tau'\}) \text{ if } (\alpha \sqcap \beta)\in C, (\forall\alpha\leqslant\tau), (\forall\beta\leqslant\tau')\in\mathcal{Q},\ \mathcal{Q} \not\vdash \alpha \mathit{\ alias\ } \beta.$$

Two universal variables are meet compatible if one is an upper bound of the other (perhaps through applications of transitivity). This is captured by the definition:

Definition 6. Define $\mathcal{Q}\vdash\alpha\leqslant\beta$ if one of the following holds:
(1) $\alpha=\beta$; or
(2) $\mathcal{Q}=\mathcal{Q}_1.\forall\alpha\leqslant\beta.\mathcal{Q}_2$; or
(3) $\mathcal{Q}\vdash\alpha\leqslant\gamma$ and $\mathcal{Q}\vdash\gamma\leqslant\beta$ for some γ.
Define $\mathcal{Q}\vdash\alpha$ *alias* β to be: $\mathcal{Q}\vdash\alpha\leqslant\beta$ or $\mathcal{Q}\vdash\beta\leqslant\alpha$.

If this upper bound condition is not satisfied, then the only other possibility is that the upper bounds for the type variables are meet compatible. This latter condition is the one checked by rule (17).

There is no corresponding rule for checking join-compatibility of universal variables, since two distinct universal variables are only join compatible if one can be promoted to the other. Given $(\beta\leqslant\alpha_1)$, $(\beta\leqslant\alpha_2)$, with β existential and α_1, α_2 universal, $\alpha_1\neq\alpha_2$, then either α_1 is an upper bound of α_1 or vice versa; otherwise there is no satisfying substitution for these two constraints.

To reason about correctness of the algorithm, we need to add the predicates $\tau \sqcap \tau'$ and $\tau \sqcup \tau'$ to the constraint logic with substitutions.

Definition 7 (*Constraint logic with* $\sqcap$ *and* $\sqcup$)**.** Extend the definition of constraint sets to include

$$C ::= \ldots \mid \{\tau \sqcap \tau'\} \mid \{\tau \sqcup \tau'\}$$

Extend the inference rules for the subtype logic in Fig. 2 (and therefore the constraint logic rules in Definition 1) with

$$\frac{\Gamma \vdash \tau \leqslant \tau'' \quad \Gamma \vdash \tau' \leqslant \tau''}{\Gamma \vdash \tau \sqcap \tau'} \quad (\textsc{SubGlb})$$

$$\frac{\Gamma \vdash \tau'' \leqslant \tau \quad \Gamma \vdash \tau'' \leqslant \tau'}{\Gamma \vdash \tau \sqcup \tau'} \quad (\textsc{SubLub})$$

Lemma 8 (Termination)**.** *The algorithm in Definition* 5 *is guaranteed to terminate.*

Proof. Termination is guaranteed by the loop check in the algorithm definition. We run the algorithm until it reaches a fixed point, taking care not to redo computations. Since all constraints contain subterms of the types in the original types, and type expressions have finite height and there are a finite number of type variables, this process is guaranteed to terminate. □

To reason about the correctness of the entailment algorithm, we use an alternative version of the subtype logic, provided in Fig. 5. In this reformulation, the SubAx and SubTrans rules are replaced by the SubProm promotion rule, restricting uses of transitivity to the application of the rule for replacing a variable with its upper bound.

Lemma 9. $\Gamma \vdash \tau \leqslant \tau'$, *in the subtype system of Fig.* 2, *if and only if* $\Gamma \vdash \tau \leqslant \tau'$ *in the normal subtype system of Fig.* 5.

$$\frac{\Gamma = \Gamma_1, \alpha \leq \tau, \Gamma_2 \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \alpha \leq \tau'} \quad (\textsc{SubProm})$$

$$\Gamma \vdash \tau \leq \tau \quad (\textsc{SubRefl})$$

$$\frac{\Gamma \vdash \tau_1' \leq \tau_1 \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash (\tau_1 \to \tau_2) \leq (\tau_1' \to \tau_2')} \quad (\textsc{SubFun})$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_1' \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash (\tau_1 * \tau_2) \leq (\tau_1' * \tau_2')} \quad (\textsc{SubPair})$$

$$\frac{\Gamma \vdash \tau_i \leq \tau_i' \text{ for } i = 1, \ldots, n \text{ where } m \geq n}{\Gamma \vdash \{\overline{l_m} : \overline{\tau_m}\} \leq \{\overline{l_n} : \overline{\tau_n'}\}} \quad (\textsc{SubRec})$$

Fig. 5. Normal subtype rules.

Proof. The proof of the "only if" part is a standard cut elimination argument, permuting uses of transitivity with uses of the rules for the type constructors. For example, given

$$\dfrac{\dfrac{\dfrac{\Pi_1}{\Gamma \vdash \tau_1 \leqslant \tau_1'} \quad \dfrac{\Pi_2}{\Gamma \vdash \tau_2 \leqslant \tau_2'}}{\Gamma \vdash (\tau_1 * \tau_2) \leqslant (\tau_1' * \tau_2')} \quad \dfrac{\dfrac{\Pi_1'}{\Gamma \vdash \tau_1' \leqslant \tau_1''} \quad \dfrac{\Pi_2'}{\Gamma \vdash \tau_2' \leqslant \tau_2''}}{\Gamma \vdash (\tau_1' * \tau_2') \leqslant (\tau_1'' * \tau_2'')}}{\Gamma \vdash (\tau_1 * \tau_2) \leqslant (\tau_1'' * \tau_2'')}$$

We transform this to the derivation

$$\dfrac{\dfrac{\dfrac{\Pi_1}{\Gamma \vdash \tau_1 \leqslant \tau_1'} \quad \dfrac{\Pi_1'}{\Gamma \vdash \tau_1' \leqslant \tau_1''}}{\Gamma \vdash \tau_1 \leqslant \tau_1''} \quad \dfrac{\dfrac{\Pi_2}{\Gamma \vdash \tau_2 \leqslant \tau_2'} \quad \dfrac{\Pi_2'}{\Gamma \vdash \tau_2' \leqslant \tau_2''}}{\Gamma \vdash \tau_2 \leqslant \tau_2''}}{\Gamma \vdash (\tau_1 * \tau_2) \leqslant (\tau_1'' * \tau_2'')}$$

We repeatedly apply transformations such as this to push uses of transitivity to the leaves of the derivation tree, stopping when the left premise of an application of transitivity comes from an application of the hypothesis rule. This process is guaranteed to terminate because type derivations have finite height. □

Lemma 10 (Soundness). *Suppose* $C \Rightarrow_{\mathscr{D}} C'$. *Then* $\theta \models \mathscr{D}C$ *if and only if* $\theta \models \mathscr{D}C'$.

Proof. The "if" part is trivial. For the "only if" part, we consider the important cases of the rule applications:

Rule (5): Assume $\theta \models \mathscr{D}C$. We need to show that if $\theta \models \mathscr{D}(\alpha \leqslant \tau')$ then $\theta \models \mathscr{D}(\tau \leqslant \tau')$. We can assume that the derivation for $\theta \models \mathscr{D}(\alpha \leqslant \tau')$ has a corresponding normal derivation, and therefore must have resulted from an application of the promotion rule. The right premise of this rule must result from a derivation for $\theta(\tau) \leqslant \theta(\tau')$.

Rules (2)–(4): Again these follow from the assumption that the original derivation is normalizable.

Rules (6)–(7): These follow from the definition of the derivability of $\tau \sqcap \tau'$ and $\tau \sqcup \tau'$ in the constraint logic.

Rules (8)–(12): These follow from the definition of the derivability of $\tau \sqcap \tau'$ and $\tau \sqcup \tau'$ in the constraint logic, and the assumption that derivations in the subtype logic are normalizable.

Rules (13)–(14): These follow from the definition of the derivability of $\tau \sqcap \tau'$ and $\tau \sqcup \tau'$ in the constraint logic.

Rule (15): By the definition of the derivability of $\alpha \sqcap \tau'$ in the constraint logic, we have derivations for $\alpha \leqslant \tau''$ and $\theta(\tau') \leqslant \tau''$ in the normal subtype system, for some τ'' and satisfying substitution θ. Since τ' is not a variable, the former derivation must follow from promoting α, and therefore there must be a derivation for $\tau \leqslant \tau''$. Therefore the constraint $\tau \sqcap \tau'$ is satisfiable.

Rule (16): By the definition of the derivability of $\tau \sqcup \tau'$ in the constraint logic, we have derivations for $\tau'' \leqslant \alpha$ and $\tau'' \leqslant \theta(\tau')$ in the normal subtype system, for some τ'' and satisfying substitution θ. Since α is a "rigid" type variable, without any hypothetical lower bound, it must be the case that $\tau'' = \beta$, for some $\beta \in UV(\mathcal{Q})$ such that $\mathcal{Q} \vdash \beta$ *alias* α. Since $\tau' \notin TyVar$, the promotion of β to α must be a subderivation of $\beta \leqslant \theta(\tau')$, and we must have a derivation for $\alpha \leqslant \theta(\tau')$. Since τ' is not a variable, and again using the normalizability of the derivation for $\alpha \leqslant \theta(\tau')$, this derivation must end with a promotion of α, with the right premise being a derivation for $\tau \leqslant \theta(\tau')$. Therefore the constraint $\tau \leqslant \tau'$ is satisfiable.

Rule (17): By the definition of the derivability of $\alpha \sqcap \beta$ in the constraint logic, we have derivations for $\alpha \leqslant \tau''$ and $\beta \leqslant \tau''$ in the normal subtype system for some τ''. Since $\mathcal{Q} \nvdash \alpha$ *alias* β, we must have $\tau'' \notin \{\alpha, \beta\}$. Therefore these derivations must end with promotions of α and β, with right premises of $\tau \leqslant \tau''$ and $\tau' \leqslant \tau''$, respectively. Therefore the constraint $\tau \sqcap \tau'$ is satisfiable. □

Lemma 11 (Containment check)**.** *Given $\mathcal{Q}$, C. Then $\mathcal{Q}C$ is not satisfiable if any of the following is contained in SUBTYPE-CLOS($\mathcal{Q}C$), but no rule of the entailment algorithm is applicable to that constraint*:

(1) $(\tau \leqslant \tau')$, $(\tau \sqcap \tau')$ *or* $(\tau \sqcup \tau') \in C$ *where* $\tau, \tau' \notin TyVar$ *and one of* τ, τ' *is not a record type*; *or*

(2) $(\alpha \leqslant \beta) \in C$ *where* $\alpha, \beta \in UV(\mathcal{Q})$, *and* $\mathcal{Q} \nvdash \alpha$ *alias* β; *or* $(\alpha \sqcap \beta)$ *or* $(\alpha \sqcup \beta) \in C$ *where* $\alpha, \beta \in UV(\mathcal{Q})$, $\mathcal{Q} \nvdash \alpha$ *alias* β; *or*

(3) $(\tau \leqslant \alpha)$, $(\alpha \sqcap \tau)$ *or* $(\alpha \sqcup \tau) \in C$ *where* $\alpha \in UV(\mathcal{Q}), \tau \notin TyVar$; *or*

(4) $(\alpha \leqslant \tau) \in C$ *where* $\mathcal{Q} = \mathcal{Q}_1 \forall \alpha \mathcal{Q}_2$ *and* $\tau \notin EV(\mathcal{Q}) \cup \{\alpha\}$.

Proof. Each step of the entailment algorithm preserves the set of satisfying substitutions. If no rule of the algorithm is applicable to the constraints of the aforesaid form, the set of constraints cannot be satisfiable, contradicting any claim that the original constraints were satisfiable. □

5. Finite satisfiability

The conditions not checked for by subtype closure are (a) finiteness of solutions, (b) scoping of type variables and (c) compatibility of the shapes of types with which a variable must be join and meet compatible. We are now going to define another form of closure for an inferred constraint set, that is used to perform checks for these conditions. The details of scope checking are provided in the next section. The details of match checking are provided in Section 7.

The subtype closure does nothing with constraints of the form $\alpha \leqslant (\alpha * \alpha)$ (for example). In systems of recursive constraints, such constraints are satisfied by solutions involving infinite trees. Since we only allow finite solutions, the existence of such constraints in the subtype closure should flag an error. The subtype relation itself is

insufficient for forming this check, consider, for example,

$$\alpha \leqslant \beta, \quad (\alpha * \alpha) \leqslant \beta.$$

We need to define a relation that is related to subtyping, but includes symmetry. Mitchell [18] and Lincoln and Mitchell [15] use a variant of the unification algorithm to check for finite satisfiability, reflecting the fact that we need to augment subtyping with symmetry to perform the check. However, they only consider subtyping between atomic types. With subtyping based on record or object interface containment, this is not sufficient. Consider these examples and their corresponding satisfying substitutions:

$$\begin{gathered}
\{x:\alpha\} \leqslant \beta, \quad \alpha \leqslant \beta,\ \theta(\alpha) = \{\,\}, \quad \theta(\beta) = \{\,\}, \\
\{x:\alpha\} \leqslant \beta,\ \ \alpha \leqslant \beta, \beta \leqslant \{x:\gamma\},\ \theta(\alpha) = \{x:\{\,\}\},\ \ \theta(\beta) = \{x:\{\,\}\},\ \ \theta(\gamma) = \{\,\}, \\
\beta \leqslant \{x:\alpha\}, \quad \beta \leqslant \alpha,\ \theta(\alpha) = \{\,\}, \quad \theta(\beta) = \{x:\{\,\}\}, \\
\{x:\alpha\} \leqslant \alpha,\ \theta(\alpha) = \{\,\}, \\
\alpha \leqslant \{x:\beta\}, \quad (\alpha * \alpha) \leqslant \beta,\ \theta(\alpha) = \{x:\{\,\} * \{\,\}\}, \quad \theta(\beta) = \{\,\} * \{\,\}.
\end{gathered}$$

On the other hand, the constraint $(\alpha \leqslant \{x:\alpha\})$ is not finitely satisfiable. Similarly, the constraint $((\{x:\alpha\} \to \mathtt{int}) \leqslant \alpha)$ is not finitely satisfiable. These examples demonstrate that we need to give special treatment to circularities in the subtype constraints involving record types. As explained in Section 1, the situation is also quite different from the situation where we have $\top$ and $\bot$ types (similarly to us, Lincoln and Mitchell omit the $\top$ and $\bot$ types). We first consider the easier case of circularities not involving record types. We define a subterm relationship based on satisfying solutions for the subtype constraints:

Definition 12 (*Match closure*). Given a quantifier prefix $\mathcal{Q}$ and constraint formula C. Define the *match closure* of $\mathcal{Q}$ and C, denoted $MATCH\text{-}CLOS(\mathcal{Q}C)$, to be the least binary relation M such that

(1) if $\mathcal{Q} = \mathcal{Q}_1.\forall\alpha \leqslant \tau.\mathcal{Q}_2$, then $(\alpha, \tau) \in M$;
(2) if $(\tau \leqslant \tau') \in SUBTYPE\text{-}CLOS(\mathcal{Q}C)$, then $(\tau, \tau') \in M$;
(3) $(\tau, \tau) \in M$;
(4) if $(\tau, \tau') \in M$ then $(\tau', \tau) \in M$;
(5) if $(\tau_1, \alpha), (\alpha, \tau_2) \in M$, then $(\tau_1, \tau_2) \in M$;
(6) if $((\tau_1 \to \tau_2), (\tau_1' \to \tau_2') \in M$, then $(\tau_1, \tau_1'), (\tau_2, \tau_2') \in M$;
(7) if $((\tau_1 * \tau_2), (\tau_1' * \tau_2')) \in M$ then $(\tau_1, \tau_1'), (\tau_2, \tau_2') \in M$.

The *strong match closure* is defined to be the match closure augmented with the following rule:

(7) if $(\{\overline{l_m} : \overline{\tau_m}\}, \{\overline{l_n'} : \overline{\tau_n'}\}) \in M$, then $(\tau_i, \tau_j') \in M$ for all i, j such that $l_i = l_j'$.

Define $\Gamma \vdash \tau \leftrightarrow \tau'$ to be: there exists a sequence of ground types $\tau_1, \ldots, \tau_{n+1}$, and $\{R_1, \ldots, R_n\} \subseteq \{\leqslant, \geqslant\}$, such that $\tau = \tau_1$, $\tau' = \tau_{n+1}$, and $\Gamma \vdash \tau_i R_i \tau_{i+1}$, for $i = 1, \ldots, n$.

Lemma 13. *If* $(\tau, \tau') \in MATCH\text{-}CLOS(\mathcal{Q}C)$, *then for all* θ *such that* $\theta \models \mathcal{Q}C$, *we have* $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\tau) \leftrightarrow \theta(\tau')$.

Proof. Given $\mathcal{Q}, C$, define $M_0 = \{(\tau, \tau') \mid (\tau \leqslant \tau') \in C\}$. Let M_i result from adding a match constraint (τ_i^1, τ_i^2) to M_{i-1} according to one of rules (3)–(6) in the definition of match closure, for $i = 1, \ldots, N$, for some N such that $M_N = MATCH\text{-}CLOS(\mathcal{Q}C)$. We verify by induction on i that $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\tau_i^1) \leftrightarrow \theta(\tau_i^2)$ for any satisfying substitution θ. Suppose that the addition of the ith constraint is justified by rule (5) (for example) in match closure. Therefore M_{i-1} must contain a constraint of the form $((\tau_1 \to \tau_2), (\tau_1' \to \tau_2'))$, where $\tau_i^1 = \tau_j$ and $\tau_i^2 = \tau_j'$ for some $j \in \{1, 2\}$. By the induction hypothesis we have $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\tau_1 \to \tau_2) \leftrightarrow \theta(\tau_1' \to \tau_2')$. We can assume that this results from a sequence of normal derivations in the subtype logic. If any two consecutive derivations in this sequence end with conclusions $\tau \geqslant \alpha$ and $\alpha \leqslant \tau'$, for some τ, τ', α, then these derivations must have ended with promotion (using the SUBPROM rule) and can be replaced by derivations with conclusions $\tau \geqslant \tau_\alpha$ and $\tau_\alpha \leqslant \tau'$, where τ_α is the upper bound of α. So we may conclude that each derivation ends with an application of SUBFUN. Therefore, from the appropriate premises in these normal derivations, we have $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\tau_j) \leftrightarrow \theta(\tau_j')$. □

Definition 14 (*Rigid dependency*)**.** Given $\mathcal{Q}$ and C, $M = MATCH\text{-}CLOS(\mathcal{Q}C)$. Define that $\alpha \in EV(\mathcal{Q})$ is rigid dependent on τ with path π, written $\mathcal{Q}C \vdash_R \alpha \xrightarrow{\pi} \tau$, if the path π does not involve record types and one of the following holds:
(1) $(\alpha, \tau') \in M$ and $\tau'(\pi) = \tau$; or
(2) $(\alpha, \tau') \in M$ and $\pi = \pi_1\pi_2$ and $\tau'(\pi_1) = \beta$, for some β, and $\mathcal{Q}C \vdash_R \beta \xrightarrow{\pi_2} \tau$.

Define that $\alpha \in EV(\mathcal{Q})$ is *strongly rigid dependent on* τ *with path* π if the above conditions hold with match closure replaced by strong match closure.

We omit record types from the definition of rigid dependency because, as explained on the previous page, for now we only considering circularities due to recursive paths in types labelled by non-record types (e.g. $\alpha \leqslant (\alpha * \alpha)$).

Lemma 15. *If* $\mathcal{Q}C \vdash_R \alpha \xrightarrow{\pi} \tau$, *then for all* θ *such that* $\theta \models \mathcal{Q}C$, $\pi \in dom(\theta(\alpha))$ *and* $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha)(\pi) \leftrightarrow \theta(\tau)$.

Proof. We reason by induction on the definition of $\mathcal{Q}C \vdash_R \alpha \xrightarrow{\pi} \tau$. For the base case, we use Lemma 13 and induction on π. For the inductive case, we use Lemma 13 and the induction hypothesis. □

To verify the occurs check, we require the following. Define the *rigid height* of a type by

$$
\begin{aligned}
&|\alpha|_{\Gamma_1,\alpha,\Gamma_2} = 0,\\
&|\alpha|_{\Gamma_1,\alpha\leqslant\tau,\Gamma_2} = |\tau|_{\Gamma_1},\\
&|(\tau_1 \to \tau_2)|_\Gamma = 1 + |\tau_1|_\Gamma + |\tau_2|_\Gamma,\\
&|(\tau_1 * \tau_2)|_\Gamma = 1 + |\tau_1|_\Gamma + |\tau_2|_\Gamma,\\
&|\{\overline{l_n} : \overline{\tau_n}\}|_\Gamma = 0.
\end{aligned}
$$

Lemma 16. *If* $\Gamma \vdash \tau_1 \leqslant \tau_2$, *then* $|\tau_1|_\Gamma = |\tau_2|_\Gamma$.

Proof. Any such derivation must be normalizable, and the result is verified by induction on the height of a normal derivation. □

For reasoning about circularities involving record types, we need a stronger condition on the occurrences of variables.

Definition 17. A path π is *positive* if the number of d symbols in the path is zero or even. The path is *negative* otherwise. Define $sign(\pi) = +$ if the path π is positive, $sign(\pi) = -$ otherwise.

Definition 18 (*Type dependency*)**.** Given $\mathcal{Q}$ and C. Define that $\alpha \in EV(\mathcal{Q})$ is type dependent on τ with path π, written $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau$, if one of the following holds:

(1)

$$\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau \text{ if } \left\{ \begin{array}{c} (\alpha \leqslant \tau') \\ (\tau' \leqslant \alpha) \end{array} \right\} \in SUBTYPE\text{-}CLOS(\mathcal{Q}C) \text{ and } \tau'(\pi) = \tau$$

and

$$\pi \text{ is } \left\{ \begin{array}{c} \text{positive} \\ \text{negative} \end{array} \right\}.$$

(2) $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau$ if $\pi = \pi_1\pi_2$, $\tau'(\pi_1) = \beta$, $\mathcal{Q}C \vdash_T \beta \xrightarrow{\pi_2} \tau$ and (a) either $(\alpha \leqslant \tau') \in SUBTYPE\text{-}CLOS(\mathcal{Q}C)$ and $sign(\pi_1) = sign(\pi_2)$, or (b) $(\tau' \leqslant \alpha) \in SUBTYPE\text{-}CLOS(\mathcal{Q}C)$ and $sign(\pi_1) \neq sign(\pi_2)$.

If $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau$, we say that α is type dependent on τ.

For example, we have the following:

$$\mathcal{Q}C \vdash_T \alpha \xrightarrow{r} \beta \quad \text{if } C \subseteq \{\alpha \leqslant (\tau \to \beta)\}, \tag{1}$$

$$\mathcal{Q}C \vdash_T \alpha \xrightarrow{d} \beta \quad \text{if } C \subseteq \{(\beta \to \tau) \leqslant \alpha\}, \tag{2}$$

$$\mathcal{Q}C \vdash_T \alpha \xrightarrow{d} \beta \quad \text{if } C \subseteq \{\alpha \leqslant (\beta \to \tau)\}, \tag{3}$$

$$\mathcal{Q}C \vdash_T \alpha \xrightarrow{dd} \beta \quad \text{if } C \subseteq \{\alpha \leqslant ((\beta \to \tau_1) \to \tau_2)\}, \tag{4}$$

$$\mathcal{Q}C \vdash_T \alpha \xrightarrow{dd} \beta \quad \text{if } C \subseteq \{\alpha \leqslant (\gamma \to \tau_2), (\beta \to \tau_1) \leqslant \gamma\}, \tag{5}$$

$$\mathcal{Q}C \vdash_T \alpha \xrightarrow{df} \beta \quad \text{if } C \subseteq \{((\beta * \tau_1) \to \tau_2) \leqslant \alpha\}, \tag{6}$$

$$\mathcal{Q}C \vdash_T \alpha \xrightarrow{df} \beta \quad \text{if } C \subseteq \{(\gamma \to \tau_2) \leqslant \alpha, \gamma \leqslant (\beta * \tau_1)\}. \tag{7}$$

Example (1) is an example of a positive dependency due to a positive occurrence of β in an upper bound, while example (2) is an example of a positive dependency due to

a negative occurrence of β in a lower bound. Example (3) is an example of a negative dependency due to a negative occurrence of β in an upper bound. Example (4) is an example of a positive dependency due to a positive occurrence of β in an upper bound, though in this case it is a negative occurrence in the domain of a function type. Example (5) is equivalent to (4), but with upper bounds "normalized" to types of depth one. Examples (6) and (7) are similar to (4) and (5) (in the sense that (7) is equivalent to (6) but with the lower bound "normalized" to types of depth one).

In the case of example (5), we have $(\alpha \leqslant (\gamma \to \tau_2)) \in C$, $(\gamma \to \tau_2)(d) = \gamma$ and $\mathcal{2}C \vdash_T \gamma \xrightarrow{d} \beta$ (and $sign(d) = (-) = sign(d)$). This corresponds to Case (a) in the third clause of the definition of type dependency. In the case of example (7), we have $(\gamma \to \tau_2) \leqslant \alpha) \in C$, $(\gamma \to \tau_2)(d) = \gamma$ and $\mathcal{2}C \vdash_T \gamma \xrightarrow{f} \beta$ (and $sign(d) = (-) \neq (+) = sign(f)$). This corresponds to case (b) in the third clause of the definition of type dependency.

Lemma 19. *If $\pi \in dom(\tau)$ then $\theta(\tau(\pi)) = \theta(\tau)(\pi)$.*

Lemma 20. *If $\Gamma \vdash \tau_1 \leqslant \tau_2$ and $\pi \in dom(\tau_1) \cap (\tau_2)$, then*

$$\left\{ \begin{array}{l} \Gamma \vdash \tau_1(\pi) \leqslant \tau_2(\pi) \\ \Gamma \vdash \tau_2(\pi) \leqslant \tau_1(\pi) \end{array} \right\}$$

if π is

$$\left\{ \begin{array}{l} \text{positive} \\ \text{negative} \end{array} \right\}.$$

Proof. By induction on the height of a normal derivation for $\Gamma \vdash \tau_1 \leqslant \tau_2$. □

Lemma 21. *If $\mathcal{2}C \vdash_T \alpha \xrightarrow{\pi} \beta$, then for all θ such that $\theta \models \mathcal{2}C$, $\pi \in dom(\theta(\alpha))$ and $UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \theta(\alpha)(\pi) \leqslant \theta(\beta)$.*

Proof. We verify this by induction on the length of the type dependency path. Assume we have

$$\mathcal{2}C \vdash_T \alpha \xrightarrow{\pi} \beta \text{ because } \pi = \pi_1\pi_2,\ \tau(\pi_1) = \gamma,\ \mathcal{2}C \vdash_T \gamma \xrightarrow{\pi_2} \beta, \text{ and}$$

(a) either $(\alpha \leqslant \tau) \in SUBTYPE\text{-}CLOS(\mathcal{2}C)$ and $sign(\pi_1) = sign(\pi_2)$,
(b) or $(\tau \leqslant \alpha) \in SUBTYPE\text{-}CLOS(\mathcal{2}C)$ and $sign(\pi_1) \neq sign(\pi_2)$.

Assume $\theta \models \mathcal{2}C$. By the induction hypothesis, we have $\pi_2 \in dom(\theta(\beta))$ and $UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \theta(\beta)(\pi_2) \leqslant \theta(\tau)$.

Suppose $(\alpha \leqslant \tau') \in SUBTYPE\text{-}CLOS(\mathcal{2}C)$, then $UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \theta(\alpha) \leqslant \theta(\tau')$. If $sign(\pi_1) = (+)$, so $sign(\pi_2) = (+)$, then using the previous lemma and the positivity of $sign(\pi_1)$, we can verify that $UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \theta(\alpha)(\pi_1) \leqslant \theta(\tau')(\pi_1) = \theta(\beta)$. Then using the previous lemma again and the positivity of $sign(\pi_2)$, we have $UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \theta(\alpha)(\pi_1\pi_2) \leqslant \theta(\beta)(\pi_2) \leqslant \theta(\tau)$. If $sign(\pi_1) = (-)$, so

$sign(\pi_2) = (-)$, then using the previous lemma and the negativity of $sign(\pi_1)$, we can verify that $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\beta) = \theta(\tau')(\pi_1) \leqslant \theta(\alpha)(\pi_1)$. Then using the previous lemma again and the negativity of $sign(\pi_2)$, we have $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha)(\pi_1\pi_2) \leqslant \theta(\beta)(\pi_2) \leqslant \theta(\tau)$.

Suppose $(\tau' \leqslant \alpha) \in SUBTYPE\text{-}CLOS(\mathcal{QC})$, then $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\tau') \leqslant \theta(\alpha)$. If $sign(\pi_1) = (-)$, so $sign(\pi_2) = (+)$, then using the previous lemma and the negativity of $sign(\pi_1)$, we can verify that $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha)(\pi_1) \leqslant \theta(\tau')(\pi_1) = \theta(\beta)$. Then using the previous lemma again and the positivity of $sign(\pi_2)$, we have $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha)(\pi_1\pi_2) \leqslant \theta(\beta)(\pi_2) \leqslant \theta(\tau)$. If $sign(\pi_1) = (+)$, so $sign(\pi_2) = (-)$, then using the previous lemma and the positivity of $sign(\pi_1)$, we can verify that $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\beta) = \theta(\tau')(\pi_1) \leqslant \theta(\alpha)(\pi_1)$. Then using the previous lemma again and the negativity of $sign(\pi_2)$, we have $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha)(\pi_1\pi_2) \leqslant \theta(\beta)(\pi_2) \leqslant \theta(\tau)$. □

Define $\mathcal{QC} \vdash \alpha \xrightarrow{\pi} \tau$ to be

(1) $\mathcal{QC} \vdash_R \alpha \xrightarrow{\pi} \tau$, or
(2) $\mathcal{QC} \vdash_T \alpha \xrightarrow{\pi} \tau$, or
(3) $\mathcal{QC} \vdash_T \alpha \xrightarrow{\pi_1} \beta$ and $\mathcal{QC} \vdash_R \beta \xrightarrow{\pi_2} \tau$, for some $\beta \in EV(\mathcal{Q})$ and $\pi = \pi_1\pi_2$.

Lemma 22 (Occurs check). *If $\mathcal{QC} \vdash \alpha \xrightarrow{\pi} \alpha$ where $\pi \neq \varepsilon$, then there is no finite substitution θ such that $\theta \models \mathcal{QC}$.*

Proof. A corollary of Lemmas 15 and 21. If $\mathcal{QC} \vdash_R \alpha \xrightarrow{\pi} \alpha$, then the path π only consists of symbols from $\{f, s, d, r\}$. Since by Lemma 15 we have $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha)(\pi) \leftrightarrow \theta(\alpha)$ for any satisfying substitution θ, there must be a sequence of derivations (and therefore normal derivations) of $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \tau_1 R_1 \tau_2, \ldots, UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \tau_n R_n \tau_{n+1}$, where $\tau_1 = \theta(\alpha)(\pi)$ and $\tau_{n+1} = \theta(\alpha)$, and $R_1, \ldots, R_n \in \{\leqslant, \geqslant\}$. By Lemma 16, $|\tau_j|_{UVARS\text{-}CONTEXT(\mathcal{Q})} = |\tau_{j+1}|_{UVARS\text{-}CONTEXT(\mathcal{Q})}$ for $j = 1, \ldots, n$. But $|\theta(\alpha)(\pi)|_{UVARS\text{-}CONTEXT(\mathcal{Q})} < |\theta(\alpha)|_{UVARS\text{-}CONTEXT(\mathcal{Q})}$, since $\pi \in \{f, s, d, r\}^*$, giving a contradiction.

If $\mathcal{QC} \vdash_T \alpha \xrightarrow{\pi} \alpha$, then by Lemma 21, we have $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha)(\pi) \leqslant \theta(\alpha)$ for any satisfying substitution θ. But then $\vdash \theta'\theta(\alpha)(\pi) \leqslant \theta'\theta(\alpha)$ for some ground satisfying substitution θ' θ, and a simple argument based on the structure of $\theta'\theta(\alpha)(\pi)$ demonstrates that this is not possible for finite types. □

Our mini-language does not allow existential variables in the upper bounds of universal variables, so for example the following constraint formula is not allowed: $\exists\alpha.\forall\beta \leqslant (\alpha * \alpha).\exists\gamma.\{\alpha \leqslant \gamma, \beta \leqslant \gamma\}$. If such constraints are allowed, then finiteness cannot be checked before entailment. Consider for example the following constraints:

$$\exists\beta.\forall\alpha \leqslant (\{x : \beta\} \to \texttt{int}).\exists\gamma.\{\alpha \leqslant (\{x : \gamma\} \to \texttt{int}),\ \beta \leqslant (\gamma * \gamma)\}.$$

6. Scoping of type variables

In this section we consider the checking of scoping of type variables in the accumulated constraints. Consider the following example:

```
letmono f = λx^β.
    let g : (∀γ.γ → γ) = λy. (if true then x else y; y)
    in (if true then x else 3; g 3; g true)
in f
```

The inner conditional requires that x and y have a common upper bound. Since γ, the type of y, ranges over all possible types, this bound must be γ, so β must be instantiable to γ. But then g cannot be polymorphic in γ. In type-checking, this error is caught by the fact that β is outside the scope of γ. "Scope" is formalized by the use of a quantifier prefix, where β is introduced to the left of γ in the prefix.

More precisely, type inference builds the quantifier prefix and constraint set

$$\exists\beta.\forall\gamma.\exists\delta.\{\beta\leqslant\delta,\ \gamma\leqslant\delta\}.$$

Any satisfying substitution for these constraints must instantiate β to γ, which is impossible since γ is introduced after β in the quantifier prefix. This dependency is detected by the match dependency: $QC\vdash\beta\xrightarrow{\varepsilon}\gamma$.

Now consider the example:

```
let f : (∀α.α → α) = λx.
    letmono g = λy^β.
        let h : (∀γ⩽α.γ → γ) = λz.
            (if true then y else z; z)
        in y
    in x
in f
```

Apparently, there is another scope violation (between β and γ in this example). However, γ has upper bound α, and β is within the scope of α. Therefore, type subsumption can be used to replace γ as a lower bound with α, effectively moving its scope out so that more existential variables are included in its scope. More formally, we have the quantified constraint set

$$\forall\alpha.\exists\beta.\forall\gamma\leqslant\alpha.\exists\delta.\{\beta\leqslant\delta,\ \gamma\leqslant\delta\}.$$

Since γ occurs positively in a lower bound, we can weaken the second constraint

$$\forall\alpha.\exists\beta.\forall\gamma\leqslant\alpha.\exists\delta.\{\beta\leqslant\delta,\ \alpha\leqslant\delta\},$$

where this step is justified by the fact that $\gamma\leqslant\delta$ is derivable from $\gamma\leqslant\alpha$ and $\alpha\leqslant\delta$.

Note that this transformation is not possible if we replace the original constraint set with

$$\forall\alpha.\exists\beta.\forall\gamma\leqslant\alpha.\exists\delta.\{\beta\leqslant\delta,(\gamma\to\gamma)\leqslant\delta\}.$$

In this example, the negative occurrence of γ in a lower bound prevents the above transformation.

For the case of unification under a mixed prefix, if an existential variable α may contain a variable β in its instantiation, the scope of the variable β is "moved out" (if necessary) so that the instantiation of α does not violate its scoping. For example, the constraint set

$$\exists\alpha.\forall\gamma.\exists\beta.\{\beta = \alpha,\ \beta = \gamma\}$$

is transformed to

$$\exists\alpha.\exists\beta.\forall\gamma.\{\beta = \alpha,\ \beta = \gamma\}.$$

At this point, the second constraint violates scoping since β's scope has moved out to the scope of α.

This movement of quantifiers cannot be done with subtyping and bounds on quantified type variables. Consider, for example,

$$\exists\alpha.\forall\gamma \leqslant \tau.\exists\beta.\{\beta \leqslant \alpha,\ \beta \leqslant \gamma\}.$$

Then $\{\gamma/\beta, \tau/\alpha\}$ is a satisfying substitution. This would not be a satisfying substitution if the scope of β were moved out to that of α.

Corollary 23 (Scope check). *$\mathcal{2}C$ is not satisfiable if there exist α and β such that either*

(1) $\mathcal{2} = \mathcal{2}_1.\exists\alpha.\mathcal{2}_2.\forall\beta \leqslant \tau.\mathcal{2}_3$ *(or* $\mathcal{2} = \mathcal{2}_1.\exists\alpha.\mathcal{2}_2.\forall\beta.\mathcal{2}_3$*) and α is type dependent on β; or*

(2) $\mathcal{2} = \mathcal{2}_1.\exists\alpha.\mathcal{2}_2.\forall\beta.\mathcal{2}_3$ *and α is rigid dependent on β.*

Proof. The verification relies on Lemmas 15 and 21 from the previous section. There we considered "false circularities" due to existential-type variables in record field types that could be omitted through record-type subsumption. Here we consider "false scope violations" due to universal-type variables that can be omitted by subsuming them with their upper bounds.

For the proof of (2), if α is rigid dependent on β with path π, then by Lemma 15 $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\alpha)(\pi) \leftrightarrow \beta$ for any satisfying substitution θ. Therefore, there is a sequence of (cut-free) derivations for $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \tau_1 R_1 \tau_2, \ldots, \mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \tau_n R_n \tau_{n+1}$, where $\tau_1 = \theta(\alpha)(\pi)$ and $\tau_{n+1} = \beta$ and $R_1, \ldots, R_n \in \{\leqslant, \geqslant\}$. Since β has no upper bound, there is no cut-free derivation for $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \beta \leqslant \tau_n$ unless $\tau_n = \beta$. The other possibility is that $\tau_n = \gamma$ and there is a derivation for $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \gamma \leqslant \beta$, where $(\forall\gamma \leqslant \beta) \in \mathcal{2}_3$. Elaborating this into an inductive argument on the length of the sequence of derivations for $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \tau_1 R_1 \tau_2, \ldots, \mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \tau_n R_n \tau_{n+1}$, we have that eventually we must have a cut-free derivation for $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\alpha)(\pi) R_1 \gamma$, for some γ universally quantified to the right of α in the quantifier prefix,

where the only upper bounds on γ are variables similarly universally quantified to the right of α. But this contradicts the scoping restriction on a satisfying substitution: any free variables in $\theta(\alpha)$ must be universally quantified to the left of α in the quantifier prefix.

For the proof of (1), if α is type dependent on β with path π, then by Lemma 21 $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha)(\pi) \leqslant \beta$ for any satisfying substitution θ. Any cut-free derivation for this requires that $\theta(\alpha)(\pi) = \beta$, which again contradicts the scoping restriction on a satisfying substitution. □

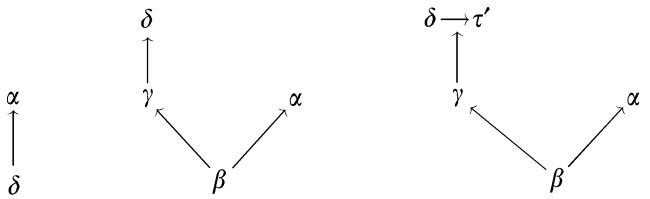
If the scope check does not fail, the subtype constraints are well scoped. The following examples illustrate the rationale for this. Consider the quantified constraint sets

$$\exists\alpha.\forall\delta \leqslant \tau.\{\delta \leqslant \alpha\},$$

$$\exists\alpha.\forall\delta \leqslant \tau.\exists\beta.\exists\gamma.\{\beta \leqslant \alpha,\ \beta \leqslant \gamma,\ \gamma \leqslant \delta\},$$

$$\exists\alpha.\forall\delta \leqslant \tau.\exists\beta.\exists\gamma.\{\beta \leqslant \alpha,\ \beta \leqslant \gamma,\ \gamma \leqslant \delta \to \tau'\}.$$

All of these are examples where $\mathcal{Q}C \vdash_R \alpha \xrightarrow{\varepsilon} \delta$, but where the dependency is not a type dependency. These examples are represented graphically as



The first of these corresponds to a positive occurrence of δ in a lower bound for α. This constraint can be instantiated by instantiating α to τ. This is not possible if δ occurs positively in an upper bound of α. The second example corresponds to $\mathcal{Q}C \vdash \alpha \xrightarrow{\varepsilon} \delta$, where the dependency path is through a path of subtype and supertype edges. The positive occurrence of δ in upper bounds for γ and β requires that these variables be instantiated to δ in any satisfying substitution. However, because the dependency of α on δ relies on the lower bound constraint ($\beta \leqslant \alpha$), it is possible to instantiate α to τ. The third example demonstrates a negative occurrence of δ in the upper bound for γ and β. In this case it is possible to instantiate all of α, β and γ to $\tau \to \tau'$.

7. Correctness of type inference

In this section we verify that a quantifier prefix $\mathcal{Q}$ and constraint set C are satisfiable if they satisfy the containment check (Lemma 11), occurs check (Lemma 22), scope check (Corollary 23), and one other check:

Lemma 24 (Match check). *$\mathcal{2C}$ is not satisfiable if $(\tau_1, \tau_2) \in MATCH\text{-}CLOS(\mathcal{2C})$ where $\tau_1, \tau_2 \notin TyVar$ have different outermost type constructors.*

Definition 25 (*Satisfying substitution*). Given $\mathcal{2C}$, define $INSTAN(\mathcal{2C})$ as follows. First, we define the set of paths in the domain of $\alpha \in EV(\mathcal{2})$, the set of universal variables that are upper bounds and the set of universal variables that are related, though not necessarily as upper bounds:

$$PATHS_{\mathcal{2C}}(\alpha) = \{\pi \mid \mathcal{2C} \vdash \alpha \xrightarrow{\pi} \tau\},$$
$$UBOUND_{\mathcal{2C}}(\alpha, \pi) = \{\beta \in UV(\mathcal{2}) \mid \mathcal{2C} \vdash_T \alpha \xrightarrow{\pi} \beta\},$$
$$RELATED_{\mathcal{2C}}(\alpha, \pi) = \{\beta \in UV(\mathcal{2}) \mid \mathcal{2C} \vdash \alpha \xrightarrow{\pi} \beta\}.$$

Define the join and meet of the upper and lower bound universal variables, respectively, constraining an existential variable:

$$\bigsqcup_{\mathcal{2C}} X = \beta \in X \quad \text{where } UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \beta \leqslant \alpha \text{ for all } \alpha \in X,$$
$$\bigsqcap_{\mathcal{2C}} X = \tau \quad \text{where } UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \alpha \leqslant \tau \text{ for all } \alpha \in X$$
$$\text{and } UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \tau \leqslant \tau'$$
$$\text{if } UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \alpha \leqslant \tau' \text{ for all } \alpha \in X.$$

To define the instantiation of a variable, we define a collection of variables, one for each path reachable from that variable. These are intuitively the nodes of a type graph representing an instantiation

$$\mathcal{2}' = \mathcal{2}.(\exists \alpha_\pi \mid \alpha \in EV(\mathcal{2}), \pi \in PATHS_{\mathcal{2C}}(\alpha)),$$
$$PATHS_{\mathcal{2C}}(\alpha_{\pi_1}) = \{\pi_2 \mid (\pi_1 \pi_2) \in PATHS_{\mathcal{2C}}(\alpha)\}.$$

Finally, the labels of the nodes in the type graph, $DEF_{\mathcal{2C}}(\alpha_\pi)$, are defined as follows:

$$\{\overline{l_k} : \overline{\alpha_{\pi l_k}}\} \quad \mathcal{2C} \vdash_T \alpha \xrightarrow{\pi l_i} \tau_i, UBOUND_{\mathcal{2C}}(\alpha, \pi) = \{\},$$
$$(\alpha_{\pi d} \to \alpha_{\pi r}) \quad \mathcal{2C} \vdash \alpha \xrightarrow{\pi} (\tau_1 \to \tau_2), UBOUND_{\mathcal{2C}}(\alpha, \pi) = \{\},$$
$$(\alpha_{\pi f} * \alpha_{\pi s}) \quad \mathcal{2C} \vdash \alpha \xrightarrow{\pi} (\tau_1 * \tau_2), UBOUND_{\mathcal{2C}}(\alpha, \pi) = \{\},$$
$$\texttt{int} \quad \mathcal{2C} \vdash \alpha \xrightarrow{\pi} \texttt{int}, UBOUND_{\mathcal{2C}}(\alpha, \pi) = \{\},$$
$$\bigsqcup_{\mathcal{2C}} X \quad X = UBOUND_{\mathcal{2C}}(\alpha, \pi) \neq \{\},$$
$$\bigsqcap_{\mathcal{2C}} X \quad X = RELATED_{\mathcal{2C}}(\alpha, \pi) \neq \{\} \text{ and}$$
$$UBOUND_{\mathcal{2C}}(\alpha, \pi) = \{\} \text{ and}$$
$$\mathcal{2C} \not\vdash \alpha \xrightarrow{\pi} \tau \text{ for } \tau \notin EV(\mathcal{2}) \cup UV(\mathcal{2}),$$
$$\{\} \quad \text{otherwise.}$$

Finally, define the solution of $\mathcal{Q}C$ to be the solution of the equation set

$$\{(\alpha = \alpha_\varepsilon), (\alpha_\pi = DEF_{\mathcal{Q}C}(\alpha_\pi)) \mid \alpha \in EV(\mathcal{Q}), \pi \in PATHS_{\mathcal{Q}C}(\alpha)\}.$$

Lemma 26. *Given that $\mathcal{Q}C$ satisfies the occurs check, $PATHS_{\mathcal{Q}C}(\alpha)$ is finite for all $\alpha \in EV(\mathcal{Q})$.*

Proof. Since the definition of paths is based on $\mathcal{Q}C \vdash \alpha \xrightarrow{\pi} \tau$, and there is a finite set of variables in $\mathcal{Q}$, an infinite set of paths $PATHS_{\mathcal{Q}C}(\alpha)$ requires cycles of the form $\mathcal{Q}C \vdash \alpha \xrightarrow{\pi} \alpha$, contradicting the fact that the occurs check is not violated. □

Lemma 27. *$DEF_{\mathcal{Q}C}(\alpha_\pi)$ is uniquely defined for all $\alpha \in EV(\mathcal{Q})$, $\pi \in PATHS_{\mathcal{Q}C}(\alpha)$.*

Proof. The match check ensures that a variable cannot be matched to types with different outermost type constructors (such as `int` and a product type). The match check also ensures that any non-variable types related to universal variables are match compatible with the non-variable upper bounds on those variables (otherwise the match check fails). The containment check ensures if a variable has variable lower bounds, and possibly also non-variable lower bounds, it has a greatest lower bound. If the variable has only variable lower bounds, it is instantiated to the greatest of these; otherwise it is instantiated to the greatest of its non-variable lower bounds, where the containment check has ensured that all variable lower bounds can be promoted to this result. The containment check also ensures that if a variable has variable upper bounds, then any non-variable upper bound is bounded below by those variable upper bounds (so the variable can be instantiated to its least upper bound). □

Lemma 28. *Let $\theta = INSTAN(\mathcal{Q}C)$, then*
(1) *$\theta(\alpha)$ is finite for all $\alpha \in EV(\mathcal{Q})$, and*
(2) *$FV(\theta(\alpha)) \subseteq UV(\mathcal{Q}_1)$ where $\mathcal{Q} = Q_1.\exists\alpha.\mathcal{Q}_2$.*

Proof. This follows from the occurs check and the scope check, respectively, and the definition of $INSTAN(\mathcal{Q}C)$. □

Lemma 29. *If*

$$\left\{ \begin{array}{l} (\alpha \leqslant \beta) \in SUBTYPE\text{-}CLOS(\mathcal{Q}C) \\ (\beta \leqslant \alpha) \in SUBTYPE\text{-}CLOS(\mathcal{Q}C) \end{array} \right\}$$

and π is

$$\left\{ \begin{array}{l} positive \\ negative \end{array} \right\},$$

then
(1) *If $\mathcal{Q}C \vdash_T \beta \xrightarrow{\pi} \tau$ then $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau$.*
(2) *$\mathcal{Q}C \vdash_R \beta \xrightarrow{\pi} \tau$ if and only if $\mathcal{Q}C \vdash_R \alpha \xrightarrow{\pi} \tau$.*

Definition 30 (*Negative-type dependency*). Given $\mathcal{Q}$ and C. Define that $\alpha \in EV(\mathcal{Q})$ is negatively type dependent on τ with path π, written $\mathcal{Q}C \vdash_{T-} \alpha \xrightarrow{\pi} \tau$, if one of the following holds:
(1)

$$\mathcal{Q}C \vdash_{T-} \alpha \xrightarrow{\pi} \tau \text{ if } \left\{ \begin{array}{l} (\tau' \leqslant \alpha) \\ (\alpha \leqslant \tau') \end{array} \right\} \in SUBTYPE\text{-}CLOS(\mathcal{Q}C)$$

and $\tau'(\pi) = \tau$ and π is

$$\left\{ \begin{array}{l} \text{positive} \\ \text{negative} \end{array} \right\}.$$

(2) $\mathcal{Q}C \vdash_{T-} \alpha \xrightarrow{\pi} \tau$ if $\pi = \pi_1\pi_2$, $\tau'(\pi_1) = \beta$, $\mathcal{Q}C \vdash_T \beta \xrightarrow{\pi_2} \tau$ and (a) either $(\alpha \leqslant \tau') \in SUBTYPE\text{-}CLOS(\mathcal{Q}C)$ and $sign(\pi_1) \neq sign(\pi_2)$, or (b) $(\tau' \leqslant \alpha) \in SUBTYPE\text{-}CLOS(\mathcal{Q}C)$ and $sign(\pi_1) = sign(\pi_2)$

If $\mathcal{Q}C \vdash_{T-} \alpha \xrightarrow{\pi} \tau$, we say that α is negatively type dependent on τ.

Lemma 31. *If $\mathcal{Q}C \vdash_{T-} \alpha \xrightarrow{\pi} \beta$, then for all θ such that $\theta \models \mathcal{Q}C$, $\pi \in dom(\theta(\alpha))$ and $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\beta) \leqslant \theta(\alpha)(\pi)$.*

Proof. Similar to the proof of Lemma 21. □

The importance of negative type dependency is given by the following.

Lemma 32. *Assume that $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau$, then*
(1) *If $\tau \equiv (\tau_1 \to \tau_2)$ then $\mathcal{Q}C \vdash_{T-} \alpha \xrightarrow{\pi d} \tau_1$ and $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi r} \tau_2$.*
(2) *If $\tau \equiv (\tau_1 * \tau_2)$ then $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi f} \tau_1$ and $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi s} \tau_2$.*
(3) *If $\tau \equiv \{l_1 : \tau_1, \ldots, l_k : \tau_k\}$ then $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi l_i} \tau_i$ for $i = 1, \ldots, k$.*
An analogous result holds if $\mathcal{Q}C \vdash_{T-} \alpha \xrightarrow{\pi} \tau$.

We verify the following by induction on τ'. It is used to verify the inductive case in Lemma 34.

Lemma 33. *Let $\theta = INSTAN(\mathcal{Q}C)$ and $FV(\tau') \subseteq UV(\mathcal{Q})$:*
(1) *Suppose $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \tau' \leqslant \tau$ for all τ such that $\mathcal{Q}C \vdash_T \alpha \xrightarrow{\pi} \tau$, then $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \tau' \leqslant \theta(\alpha_\pi)$.*
(2) *Suppose $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \tau \leqslant \tau'$ for all τ such that $\mathcal{Q}C \vdash_{T-} \alpha \xrightarrow{\pi} \tau$, then $UVARS\text{-}CONTEXT(\mathcal{Q}) \vdash \theta(\alpha_\pi) \leqslant \tau'$.*

Lemma 34. *If $(\alpha \leqslant \beta) \in SUBTYPE\text{-}CLOS(\mathcal{Q}C)$ and $\theta = INSTAN(\mathcal{Q}C)$ and $\pi \in PATHS_{\mathcal{Q}C}(\alpha) \cap PATHS_{\mathcal{Q}C}(\beta)$, then*

$$\left\{ \begin{array}{l} UVARS\text{-}CONTEXT(\mathcal{Q}C) \vdash \theta(\alpha_\pi) \leqslant \theta(\beta_\pi) \\ UVARS\text{-}CONTEXT(\mathcal{Q}C) \vdash \theta(\beta_\pi) \leqslant \theta(\alpha_\pi) \end{array} \right\}$$

if π *is*

$$\left\{ \begin{array}{l} positive \\ negative \end{array} \right\}.$$

Proof. By induction on $\max(|\theta(\alpha_\pi)|, |\theta(\beta_\pi)|)$, using Lemma 29. For the base case, assume that π is positive, then there are two cases to consider:

$\mathcal{2C} \vdash_T \beta \xrightarrow{\pi} \tau$, i.e., $\pi \in PATHS_{\mathcal{2C}}(\beta)$ because of a type dependency. If $|\theta(\beta_\pi)| = 0$ then $\theta(\beta_\pi) \in \{\mathtt{int}, \{\}\} \cup UV(\mathcal{2})$. If $\theta(\beta_\pi) = \mathtt{int}$ then $\theta(\alpha_\pi) = \mathtt{int}$ or $\theta(\alpha_\pi) \in UV(\mathcal{2})$ is promotable to `int` (by the containment check and Lemma 27). If $\theta(\beta_\pi) \in UV(\mathcal{2})$ then $\theta(\alpha_\pi) \in UV(\mathcal{2})$ is promotable to $\theta(\beta_\pi)$ (by the containment check and the definition of $\sqcup_{\mathcal{2C}} UBOUND_{\mathcal{2C}}(\alpha, \pi)$ and Lemma 27). If $\theta(\beta_\pi) = \{\}$ then by the containment check and Lemma 27, $\theta(\alpha_\pi)$ is also a record type. If $|\theta(\alpha_\pi)| = 0$ then $|\theta(\beta_\pi)| = 0$ and the previous analysis applies.

$\mathcal{2C} \vdash_R \beta \xrightarrow{\pi} \tau$, i.e., $\pi \in PATHS_{\mathcal{2C}}(\beta)$ because of a rigid dependency. If any but the default case in the definition of $DEF_{\mathcal{2C}}(\beta)$ applies, then this case also applies to the definition of $DEF_{\mathcal{2C}}(\beta)$. If the default case applies in the definition of $DEF_{\mathcal{2C}}(\beta)$ but does not apply in the case of $DEF_{\mathcal{2C}}(\alpha)$, this can only happen because $\mathcal{2C} \vdash_T \alpha \xrightarrow{\pi} \{\ldots\}$, i.e., because of a type dependency of α on a record field, but then $\theta(\beta_\pi) = \{\}$ is a supertype of $\theta(\alpha_\pi)$.

For the inductive case, assume that π is positive, then there are again two cases to consider:

$\mathcal{2C} \vdash_T \beta \xrightarrow{\pi} \tau$ i.e., $\pi \in PATHS_{\mathcal{2C}}(\beta)$ because of a type dependency. Suppose for example that $\theta(\beta_\pi) = (\theta(\beta_{\pi d}) \to \theta(\beta_{\pi r}))$, therefore we must have either $\theta(\alpha_\pi) = \gamma \in UV(\mathcal{2})$ or $\theta(\alpha_\pi) = (\theta(\alpha_{\pi d}) \to \theta(\alpha_{\pi r}))$ (i.e., the second case in the definition of $DEF_{\mathcal{2C}}(\alpha_\pi)$ applies). In the former case, if $\mathcal{2C} \vdash_T \beta \xrightarrow{\pi} (\tau_1 \to \tau_2)$, then also $\mathcal{2C} \vdash_T \alpha \xrightarrow{\pi} (\tau_1 \to \tau_2)$. If $\mathcal{2C} \vdash_T \alpha \xrightarrow{\pi} (\tau_1 \to \tau_2)$ and $\mathcal{2} = \mathcal{2}_1.\forall\gamma \leqslant (\tau_1' \to \tau_2').\mathcal{2}_2$, then subtype closure must generate the sequence of constraints

$$\gamma \sqcup (\tau_1 \to \tau_2), \quad (\tau_1' \to \tau_2') \leqslant (\tau_1 \to \tau_2), \quad \tau_1 \leqslant \tau_1', \quad \tau_2' \leqslant \tau_2$$

Using Lemma 33 we therefore have that $UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \theta(\alpha_{\pi d}) \leqslant \tau_1'$ and $UVARS\text{-}CONTEXT(\mathcal{2}) \vdash \tau_2' \leqslant \theta(\alpha_{\pi r})$. By the induction hypothesis we have $\theta(\beta_{\pi d}) \leqslant \theta(\alpha_{\pi d})$ (since πd is negative) and $\theta(\alpha_{\pi d}) \leqslant \theta(\beta_{\pi d})$, so we conclude that

$$\theta(\alpha_\pi) = \gamma \leqslant (\tau_1' \to \tau_2') \leqslant (\theta(\alpha_{\pi d}) \to \theta(\alpha_{\pi r})) \leqslant (\theta(\beta_{\pi d}) \to \theta(\beta_{\pi r}))$$

If $\theta(\alpha_\pi) = (\theta(\alpha_{\pi d}) \to \theta(\alpha_{\pi r}))$, then we apply the induction hypothesis to conclude $\theta(\beta_{\pi d}) \leqslant \theta(\alpha_{\pi d})$ and $\theta(\alpha_{\pi d}) \leqslant \theta(\beta_{\pi d})$, and the result follows. The other cases are analogous.

$\mathcal{2C} \vdash_R \beta \xrightarrow{\pi} \tau$ i.e., $\pi \in PATHS_{\mathcal{2C}}(\beta)$ because of a rigid dependency. Similar to the base case, but using the induction hypothesis for matched subterms.

For the case where π is negative, the reasoning is similar, although transposing α_π and β_π. □

Corollary 35. *If* $(\alpha \leqslant \beta) \in \mathit{SUBTYPE\text{-}CLOS}(\mathcal{2C})$ *and* $\theta = \mathit{INSTAN}(\mathcal{2C})$, *then* $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2C}) \vdash \theta(\alpha) \leqslant \theta(\beta)$.

Lemma 36. *Suppose* $\theta = \mathit{INSTAN}(\mathcal{2C})$.
(1) *If* $(\alpha \leqslant \tau) \in \mathit{SUBTYPE\text{-}CLOS}(\mathcal{2C})$ *and* $\beta = \tau(\pi)$, *then*

$$\left\{ \begin{array}{l} \mathit{UVARS\text{-}CONTEXT}(\mathcal{2C}) \vdash \theta(\alpha)(\pi) \leqslant \theta(\beta) \\ \mathit{UVARS\text{-}CONTEXT}(\mathcal{2C}) \vdash \theta(\beta) \leqslant \theta(\alpha)(\pi) \end{array} \right\} \quad \textit{if the path } \pi \textit{ is} \quad \left\{ \begin{array}{l} \textit{positive} \\ \textit{negative} \end{array} \right\}.$$

(2) *If* $(\tau \leqslant \alpha) \in \mathit{SUBTYPE\text{-}CLOS}(\mathcal{2C})$ *and* $\beta = \tau(\pi)$, *then*

$$\left\{ \begin{array}{l} \mathit{UVARS\text{-}CONTEXT}(\mathcal{2C}) \vdash \theta(\beta) \leqslant \theta(\alpha)(\pi) \\ \mathit{UVARS\text{-}CONTEXT}(\mathcal{2C}) \vdash \theta(\alpha)(\pi) \leqslant \theta(\beta) \end{array} \right\} \quad \textit{if the path } \pi \textit{ is} \quad \left\{ \begin{array}{l} \textit{positive} \\ \textit{negative} \end{array} \right\}.$$

Proof. By induction on π, using Corollary 35 for the base case. □

Lemma 37. *If* $\mathcal{2C} \vdash_R \alpha \xrightarrow{\pi} \tau$ *and* $\theta = \mathit{INSTAN}(\mathcal{2C})$, *then* $\pi \in \mathit{dom}(\theta(\alpha))$ *and* $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\alpha)(\pi) \leftrightarrow \theta(\tau)$.

Proof. Similar to the proof of Lemma 15, by induction on π. □

Lemma 38. *Suppose* $\theta = \mathit{INSTAN}(\mathcal{2C})$:

(1) If $\mathcal{2C} \vdash_T \alpha \xrightarrow{\pi} \tau$, *then* $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\alpha)(\pi) \leqslant \theta(\tau)$.
(2) If $\mathcal{2C} \vdash_{T-} \alpha \xrightarrow{\pi} \tau$, *then* $\mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\tau) \leqslant \theta(\alpha)(\pi)$.

Proof. Similar to the proof of Lemma 21, by induction on π and using Lemma 36. □

Theorem 39. *Let* $C' = \mathit{SUBTYPE\text{-}CLOS}(\mathcal{2C})$ *and* $\theta = \mathit{INSTAN}(\mathcal{2C})$. *Then*

$$\left\{ \begin{array}{l} \mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\alpha) \leqslant \theta(\tau) \\ \mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\tau) \leqslant \theta(\alpha) \\ \mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\tau) \sqcup \theta(\alpha) \\ \mathit{UVARS\text{-}CONTEXT}(\mathcal{2}) \vdash \theta(\tau) \sqcap \theta(\alpha) \end{array} \right\} \textit{ if } \left\{ \begin{array}{l} (\alpha \leqslant \tau) \in C' \\ (\tau \leqslant \alpha) \in C' \\ (\tau \sqcup \alpha) \in C' \\ (\tau \sqcap \alpha) \in C' \end{array} \right\}.$$

Proof. By induction on $\max(|\theta(\alpha)|, |\theta(\tau)|)$, using the previous lemma. □

Corollary 40. *If* $\mathcal{2C}$ *does not violate the containment check, occurs check, scope check or match check, then* $\models \mathcal{2C}$.

Lemma 41. *Given an execution of the type inference algorithm, and an application of* Let *in this execution that introduces universal variables* $\{\overline{\alpha}\}$, *with outputs* $\mathcal{2}^{\mathrm{local}}$

and C^{local}, *while type-checking the definition in a* `let`. *Let* $\mathscr{Q}^{\text{global}}$ *and* C^{global} *be the global outputs of the algorithm. Then for any* $\beta \in EV(\mathscr{Q}^{\text{global}})$, $\alpha \in \{\overline{\alpha}\}$, *we have* $\mathscr{Q}^{\text{local}}C^{\text{local}} \vdash \beta \xrightarrow{\pi} \alpha$ *if and only if* $\mathscr{Q}^{\text{global}}C^{\text{global}} \vdash \beta \xrightarrow{\pi} \alpha$.

Proof. Modify the execution of the type inference algorithm as follows. Assume two distinct classes of variables, *DISCHARGED* and *UNDISCHARGED*. Define the mapping $INSCOPE(\alpha) \subseteq EV(\mathscr{Q})$ for $\alpha \in DISCHARGED \cap UV(\mathscr{Q})$ as follows, by induction on an execution of the type inference algorithm. Define the execution of the type inference algorithm on an expression $e \equiv \texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2$ where $\sigma \equiv \forall\overline{\alpha_k} \leqslant \overline{\tau'_k}.\tau_1$. Let $\mathscr{Q}_0 = \mathscr{Q}\forall\overline{\alpha_k} \leqslant \overline{\tau'_k}$, where $\{\alpha_1, \ldots, \alpha_k\} \subseteq UNDISCHARGED$. Suppose $\mathscr{Q}_0; A \vdash e_1 : \tau_1$ *with* $\mathscr{Q}_1, C_1$, let $\mathscr{Q}'_1 = \mathscr{Q}_L.\forall\overline{\beta_k} \leqslant \overline{\{\overline{\beta_k}/\overline{\alpha_k}\}\tau'_k}.\mathscr{Q}_R$ where $\mathscr{Q}_1 = \mathscr{Q}_L.\forall\overline{\beta_k} \leqslant \overline{\{\overline{\beta_k}/\overline{\alpha_k}\}\tau'_k}.\mathscr{Q}_R$ and $\{\beta_1, \ldots, \beta_k\} \subseteq (DISCHARGED - (EV(\mathscr{Q}_1) \cup UV(\mathscr{Q})))$. Define $INSCOPE(\beta_i) = EV(\mathscr{Q}'_1) - EV(\mathscr{Q})$, the existential variables introduced while α_i was undischarged. Then $\mathscr{Q}'_1; A, x : \sigma' \vdash e_2 : \tau_2$ *with* $\mathscr{Q}_2$, C_2 where $\sigma' \equiv \forall\overline{\beta_k} \leqslant \overline{\{\overline{\beta_k}/\overline{\alpha_k}\}\tau'_k}.\{\overline{\beta_k}/\overline{\alpha_k}\}\tau$. Then the output of the algorithms are the quantifier prefix $\mathscr{Q}_2$, the type τ_2 and the constraint set $\{\overline{\beta_k}/\overline{\alpha_k}\}C_1 \cup C_2$. The following claim is verified by induction on the execution of the type inference algorithm:

If $\mathscr{Q}; A \vdash e : \tau$ *with* $\mathscr{Q}'$, C and $V = FV(A) \cup FV(\tau)$, then
(1) $V \cap DISCHARGED \cap UV(\mathscr{Q}) = \{\}$, and
(2) $V \cap \bigcup\{INSCOPE(\alpha) \mid \alpha \in DISCHARGED \cap UV(\mathscr{Q})\} = \{\}$.

Now, let $\mathscr{Q}^{\text{local}}, C^{\text{local}}, \mathscr{Q}^{\text{global}}, C^{\text{global}}, \alpha, \beta$ be as in the statement of the lemma ($\alpha \in DISCHARGED \cap UV(\mathscr{Q})$). Suppose $\mathscr{Q}^{\text{local}}C^{\text{local}} \not\vdash \beta \to \alpha$ and $\mathscr{Q}^{\text{global}}\mathscr{C}^{\text{global}} \vdash \beta \xrightarrow{\pi} \alpha$. This must be due to the addition of a subtype constraint in the subsequent execution of the type inference algorithm. Because of the aforesaid claim, this additional constraint must only involve variables involving $UNDISCHARGED \cap UV(\mathscr{Q})$ and $EV(\mathscr{Q}) - \bigcup\{INSCOPE(\alpha) \mid \alpha \in DISCHARGED \cap UV(\mathscr{Q})\}$, i.e., undischarged universal variables and existential variables out of the scope of discharged variables. Because the let-definition type does not have any free variables, any constraints involving α must be generated while type-checking the let-definition, and these constraints can only involve existential variables $\beta \in EV(\mathscr{Q}^{\text{local}})$, where $INSCOPE(\alpha) \subseteq EV(\mathscr{Q}^{\text{local}})$. The constraint(s) that subsequently lead to $\mathscr{Q}^{\text{global}}C^{\text{global}} \vdash \gamma \xrightarrow{\pi} \alpha$ cannot involve $\beta \in INSCOPE(\alpha)$, because these variables are not free in A or σ. Therefore, $\mathscr{Q}^{\text{global}}C^{\text{global}} \vdash \gamma \xrightarrow{\pi} \alpha$ must be due to constraints that involve $\beta \in (EV(\mathscr{Q}^{\text{local}}) - INSCOPE(\alpha))$. But then an induction on the length π verifies that this leads to a scope violation in the final constraints (since all of these existential variables are to the left of α in the quantifier prefix, having been introduced before the let-definition was type-checked). □

We verify the following by induction on the execution of the type inference algorithm, using Corollary 40 and Lemma 41.

Theorem 42 (Soundness of type inference). *Given expression e, type environment A, and quantifier prefix $\mathcal{Q}$. Suppose $\mathcal{Q}; A \vdash e : \tau$ with $\mathcal{Q}'$, C and $\models \mathcal{Q}'C$. Let θ be the satisfying substitution computed in the proof of Corollary* 40. *Then $UVARS\text{-}CONTEXT(\mathcal{Q}'); \theta(A) \vdash e : \theta(\tau)$.*

Theorem 43 (Completeness of type inference). *Given $\Gamma \subseteq UVARS\text{-}CONTEXT(\mathcal{Q})$ and $dom(\theta) \subseteq EV(\mathcal{Q})$. Given $\Gamma; \theta(A) \vdash e : \tau$, and $\mathcal{Q}; A \vdash e : \tau'$ with $\mathcal{Q}'$, C. Then there exists θ', θ'' such that $\theta' = \theta''\,\theta$ and $\theta' \models (\mathcal{Q}'C)$ and $\Gamma \vdash \theta'(\tau') \leqslant \tau$.*

Proof. The substitution θ in the statement of the theorem is used to strengthen the induction hypothesis, to push the induction through. We first verify that, for any type derivation, there is a canonical-type derivation where there is exactly one application of SUB following an application of a structural rule. We then use induction on the height of the given canonical-type derivation to verify the theorem. We consider the key cases (from Fig. 1 in Section 2):

VAR: We have $A(x) = \sigma = \forall \overline{\alpha_n} \leqslant \overline{\tau_n}\ .\tau$ and an application of VAR followed by an application of SUB:

$$\dfrac{\dfrac{\theta(A)(x) = \theta(\sigma) = \forall \overline{\alpha_n} \leqslant \overline{\theta(\tau_n)}.\theta(\tau) \quad \Gamma \vdash \overline{\tau'_n} \leqslant \overline{\theta(\tau_n)}}{\Gamma; \theta(A) \vdash x : \{\overline{\tau'_n}/\overline{\alpha_n}\}\theta(\tau)} \qquad \Gamma \vdash \{\overline{\tau'_n}/\overline{\alpha_n}\}\theta(\tau) \leqslant \tau''}{\Gamma; \theta(A) \vdash x : \tau''}$$

Since $\Gamma \vdash \overline{\tau'_n} \leqslant \overline{\theta(\tau_n)}$ is a premise to the application of the VAR rule, let $\theta'' = \{\overline{\tau'_n}/\overline{\alpha_n}\}$, where $\{\overline{\alpha_n}\}$ are the new existential type variables introduced by the VAR step in the type inference algorithm ($\mathcal{Q}' = \mathcal{Q}\exists\overline{\alpha_n}$). Then $\theta''\theta \models \mathcal{Q}'\{\overline{\alpha_n} \leqslant \overline{\tau_n}\}$, since $UVARS\text{-}CONTEXT(\mathcal{Q}') \vdash \overline{\theta''(\alpha_n)} \leqslant \overline{\theta(\tau_n)}$, and $\Gamma \vdash \{\overline{\tau'_n}/\overline{\alpha_n}\}\theta(\tau) \leqslant \tau''$ *where* $\{\overline{\tau'_n}\}/\{\overline{\alpha_n}\}$ $\theta(\tau) = \theta''\theta(\tau)$.

APP: The rator has a derivation for type $\tau_2 \to \tau_1$, and the rand has a derivation for type τ_2.

$$\dfrac{\dfrac{\Gamma; \theta(A) \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma; \theta(A) \vdash e_2 : \tau_2}{\Gamma; \theta(A) \vdash e_1(e_2) : \tau_1} \qquad \Gamma \vdash \tau_1 \leqslant \tau}{\Gamma; \theta(A) \vdash e_1(e_2) : \tau}$$

The type inference algorithm infers the types τ'_1 and τ'_2 for e_1 and e_2, respectively, and adds the constraint $\tau'_1 \leqslant (\tau'_2 \to \alpha)$, where α is new. By the induction hypothesis there are substitutions θ''_1 and θ''_2 such that $\Gamma \vdash \theta''_1\theta(\tau'_1) \leqslant (\tau_2 \to \tau_1)$ and $\Gamma \vdash \theta''_2\theta(\tau'_2) \leqslant \tau_2$. Choose $\theta' = \{\tau_2/\alpha\}\ \theta''_1\ \theta''_2\ \theta$. Then we deduce $\Gamma \vdash \theta'(\tau'_1) \leqslant \tau_2 \to \tau_1 \leqslant \theta'(\tau'_2) \to \tau_1 = \theta'(\tau'_2 \to \alpha)$. □

8. Decidable semantic entailment

Smith and Trifonov [29] define a semantic order on polymorphic constrained types. The usefulness of this relation is in matching the types in an implementation module with the types required in an interface for that module. Although the subtype rules and algorithms presented earlier can be used for this purpose, the semantic ordering gives a more flexible ordering, based on an extensional consideration of all possible instantiations of polymorphic types. We must then consider the relationship between the earlier type system and this semantic ordering. For simplicity, we only work with constrained types for closed expressions (with no free variables).

Definition 44 (*Instances of polymorphic types*)**.** An *instance* of a polymorphic type $\forall \overline{\alpha_m} \leqslant \overline{\tau_m}.\tau$ is the ground type $\theta(\tau)$, for an assignment θ that satisfies the constraints in the hypotheses: $\theta \models \exists \overline{\alpha_m}.\overline{\alpha_m} \leqslant \overline{\tau_m}$.

A decision procedure for the semantic ordering of polymorphic types, in the general case of (potentially infinite) regular trees, remains an open problem. We now demonstrate that this relation is decidable if the interpretation of types is restricted to finite trees.

Definition 45 (*Ordering of polymorphic types*)**.** Define a semantic ordering on polymorphic types by $\sigma \leqslant^{\forall}_{sem} \sigma'$ if for each instance of σ' there is a smaller instance of σ.

We need the following notion of semantic entailment:

$$\theta \models \Gamma \quad \text{if} \quad dom(\theta) = dom(\Gamma) \text{ and } \vdash \theta(\alpha) \leqslant \theta(\tau) \text{ for all } (\alpha \leqslant \tau) \in \Gamma$$

$$\Gamma \models \tau_1 \leqslant \tau_2 \quad \text{if} \quad \forall \theta.(\theta \models \Gamma) \text{ implies } \vdash \theta(\tau_1) \leqslant \theta(\tau_2)$$

The subtype rules of Fig. 2 are easily shown to be sound with respect to semantic entailment, however they are not complete. The difficulty is with the case where a subtype bound only admits a single type. Define a singleton type as

> A type τ is a *singleton type* if the only occurrence of a record type in τ is the empty record type in negative position, and any free variables in τ are constrained by upper bounds that are singleton types.

To make this inference system complete with respect to the aforesaid semantic model, we add the following inference rule:

$$\frac{\Gamma = \Gamma_1, \alpha \leqslant \tau, \Gamma_2 \quad \tau \text{ is a singleton type}}{\Gamma \vdash \tau \leqslant \alpha} \qquad (\textsc{SubSing})$$

Intuitively, subtype upper bounds of the form $(\alpha \leqslant \texttt{int})$ and $(\alpha \leqslant (\texttt{int} * \texttt{int} \to \texttt{int}))$ only allow one possible instantiation for the type variable α. A more subtle constraint is one of the form $(\alpha \leqslant (\{\} \to \texttt{int}))$. On the other hand, β is not constrained by a singleton type in the context $\{\alpha \leqslant \{\}, \beta \leqslant (\alpha \to \texttt{int})\}$.

We also must adapt the constraint-checking algorithms to the situation where universal-type variables may implicitly have lower bounds (the same as their upper bounds, in the case where the upper bound is a singleton type). The additions to the entailment algorithm of Fig. 4 are the following rules:

$$C \Rightarrow_{\mathscr{Q}} (C \cup \{\tau' \leqslant \tau\}) \text{ if } \mathscr{Q} = \mathscr{Q}_1 \forall \alpha \leqslant \tau \mathscr{Q}_2 \text{ and}$$
$$(\tau' \leqslant \alpha) \in C,\ \tau' \notin EV(\mathscr{Q}),\ \tau \text{ is a singleton type}$$
$$C \Rightarrow_{\mathscr{Q}} (C \cup \{\tau \sqcup \tau'\}) \text{ if } \mathscr{Q} = \mathscr{Q}_1 \forall \alpha \leqslant \tau \mathscr{Q}_2 \text{ and}$$
$$(\alpha \sqcup \tau') \in C,\ \tau' \notin TyVar,\ \tau \text{ is a singleton type}$$

The first of these rules complements rule (5). The second of these rules takes precedence over rule (16).

The statement of the scope check is modified as follows:

Corollary 46 (Scope check). *$\mathscr{Q}C$ is not satisfiable if there exist α and β such that either*

(1) *$\mathscr{Q} = \mathscr{Q}_1.\exists\alpha.\mathscr{Q}_2.\forall\beta \leqslant \tau.\mathscr{Q}_3$ (or $\mathscr{Q} = \mathscr{Q}_1.\exists\alpha.\mathscr{Q}_2.\forall\beta.\mathscr{Q}_3$) and α is type dependent on β, and τ is not a singleton type; or*
(2) *$\mathscr{Q} = \mathscr{Q}_1.\exists\alpha.\mathscr{Q}_2.\forall\beta.\mathscr{Q}_3$ and α is rigid dependent on β.*

The verification of the modified constraint-checking algorithms is similar to the original verification. We now concentrate on the completeness of the subtype logic with respect to semantic entailment.

Theorem 47. *$\Gamma \vdash \tau_1 \leqslant \tau_2$ if and only if $\Gamma \models \tau_1 \leqslant \tau_2$.*

Proof. Soundness (the "only if" part) is verified by induction on derivations in the subtype logic. Completeness (the "if" part) is verified by contraposition. Assume $\Gamma \nvdash \tau_1 \leqslant \tau_2$, then we show $\Gamma \not\models \tau_1 \leqslant \tau_2$. We construct a θ such that $\theta \models \Gamma$ and $\nvdash \theta(\tau_1) \leqslant \theta(\tau_2)$. By an argument based on normalizability of derivations, we may assume that derivability fails due to the non-derivability of a judgement of the form

(1) $\Gamma \vdash \tau \leqslant \alpha$ where $\tau \notin TyVar$ and α does not have a singleton upper bound in Γ;
(2) $\Gamma \vdash \alpha \leqslant \beta$; or
(3) $\Gamma \vdash \alpha \leqslant \tau$ where $\tau \notin TyVar$ and α has no upper bound in Γ.

For all cases where α (or β) has no upper bound in Γ, we may choose a substitution θ distinguishing that variable from its bound in the judgement. The remaining interesting cases are:

(1) $\Gamma \vdash \tau \leqslant \beta$ where $\tau \notin TyVar$ and β has an upper bound τ_β; or
(2) $\Gamma \vdash \alpha \leqslant \beta$ where α and β have upper bounds τ_α and τ_β, respectively.

We only consider case (2), since case (1) is similar. Assume $\Gamma \nvdash \alpha \leqslant \beta$. Assume Γ is the sequence $\Gamma_1, \ldots, \Gamma_n$, where each Γ_i is either α_i or $\alpha_i \leqslant \tau_i$. Let $\mathscr{L}$ be a collection of

record field labels that do not occur in any of the original types; let $\uplus$ denote disjoint union. We construct the substitution θ as the composition $\theta_n\ \theta_{n-1}\ \cdots\ \theta_1$, where:

- If $\alpha_i = \beta$, then $\theta_i(\beta) = LB_\perp(L_i, \theta_{i-1} \cdots \theta_1(\tau_i))$.
- Otherwise if α_i has no upper bound, define $\theta(\alpha_i) = \{l : \mathtt{int}\}$, where $L_i = \{l\}$.
- Otherwise if $\alpha_i \neq \beta$ has the upper bound τ_i, then set $\theta_i(\alpha_i) = UB_\top(L_i, \theta_{i-1} \cdots \theta_1(\tau_i))$,

here $\mathscr{L} = \biguplus_{i=1}^{n} L_i$, and where the metafunctions $LB_\perp(L,\tau)$, $LB_\top(L,\tau)$, $UB_\perp(L,\tau)$ and $UB_\top(L,\tau)$ are defined as follows:

$$
\begin{aligned}
UB_\top(\{\}, \mathtt{int}) &= \mathtt{int},\\
UB_\top(L_1 \uplus L_2, \tau_1 \to \tau_2) &= LB_\top(L_1,\tau_1) \to UB_\top(L_2,\tau_2),\\
UB_\top(L_1 \uplus L_2, \tau_1 * \tau_2) &= UB_\top(L_1,\tau_1) * UB_\top(L_2,\tau_2),\\
UB_\top\left(\left(\biguplus_{i=1}^{m} L_i\right) \uplus \{l\}, \{\overline{l_m} : \overline{\tau_m}\}\right) &= \{\overline{l_m} : \overline{UB_\top(L_m,\tau_m)}, l : \mathtt{int}\},\\
LB_\top(\{\}, \mathtt{int}) &= \mathtt{int},\\
LB_\top(L_1 \uplus L_2, \tau_1 \to \tau_2) &= UB_\top(L_1,\tau_1) \to LB_\top(L_2,\tau_2),\\
LB_\top(L_1 \uplus L_2, \tau_1 * \tau_2) &= LB_\top(L_1,\tau_1) * LB_\top(L_2,\tau_2),\\
LB_\top\left(\left(\biguplus_{i=1}^{m} L_i\right), \{\overline{l_m} : \overline{\tau_m}\}\right) &= \{\overline{l_m} : \overline{LB_\top(L_m,\tau_m)}\},\\
UB_\perp(\{\}, \mathtt{int}) &= \mathtt{int},\\
UB_\perp(L_1 \uplus L_2, \tau_1 \to \tau_2) &= LB_\perp(L_1,\tau_1) \to UB_\perp(L_2,\tau_2),\\
UB_\perp(L_1 \uplus L_2, \tau_1 * \tau_2) &= UB_\perp(L_1,\tau_1) * UB_\perp(L_2,\tau_2),\\
UB_\perp\left(\left(\biguplus_{i=1}^{m} L_i\right) \uplus \{l\}, \{\overline{l_m} : \overline{\tau_m}\}\right) &= \{\overline{l_m} : \overline{UB_\perp(L_m,\tau_m)}, l : \mathtt{int}\},\\
LB_\perp(\{\}, \mathtt{int}) &= \mathtt{int},\\
LB_\perp(L_1 \uplus L_2, \tau_1 \to \tau_2) &= UB_\perp(L_1,\tau_1) \to LB_\perp(L_2,\tau_2),\\
LB_\perp(L_1 \uplus L_2, \tau_1 * \tau_2) &= LB_\perp(L_1,\tau_1) * LB_\perp(L_2,\tau_2),\\
LB_\perp(\{\}, \{\overline{l_m} : \overline{\tau_m}\}) &= \{\},
\end{aligned}
$$

where $\mathscr{L}$ and the partitioning $L_1, \ldots, L_n$ of $\mathscr{L}$ is chosen to make this function well defined.

We now claim that $\nvdash \theta(\alpha) \leqslant \theta(\beta)$. Since α and β have upper bounds, we may assume that (or give an inductive argument to show that) their instantiations contain record types (because we disallow singleton types as upper bounds in Γ). Assume $\vdash \theta(\alpha) \leqslant \theta(\beta)$, and let π be a positive path such that $\theta(\alpha)(\pi)$ and $\theta(\beta)(\pi)$ are record types. Then $l \in dom(\theta(\beta)(\pi))$, where $l \in \mathscr{L}$ is introduced by $UB_\perp(\ldots,\ldots)$ when computing the instantiation of β, but $l \notin dom(\theta(\beta)(\pi))$. The argument for this is that $\Gamma \nvdash \alpha \leqslant \beta$, so the only other way to have a record type with field l in $\theta(\alpha)$ is if there

is a sequence of variables $\gamma_1,\ldots,\gamma_{k+1}$ and types $\tau'_1,\ldots,\tau'_k$ such that $\alpha=\gamma_1$, $\gamma_{k+1}=\beta$ and $(\gamma_i\leqslant\tau'_i)\in\Gamma$, $\gamma_{i+1}\in FV(\tau'_i)$ for $i=1,\ldots,k$ (where $\tau'_i\notin TyVar$ for some i). But then any path in $\theta(\alpha)$ to a record type containing l must consist of a path in $\theta(\beta)$ prefixed by a non-empty path π', contradicting the assertion that both $\theta(\alpha)$ and $\theta(\beta)$ have a path π to a record type with the l label.

Alternatively assume $\vdash\theta(\alpha)\leqslant\theta(\beta)$, and let π be a negative path such that $\theta(\alpha)(\pi)$ and $\theta(\beta)(\pi)$ are record types. Then $\theta(\beta)(\pi)=\{\}$ but $\theta(\alpha)(\pi)$ must contain at least one field label (perhaps the label l that is inserted by $UB_\top(\ldots,\ldots)$ to ensure there is at least one such label). If the only record types in $\theta(\alpha)(\pi)$ are empty record types in negative position, then the upper bound of α is a singleton type, contradicting the assertion that $\Gamma\nvdash\alpha\leqslant\beta$. An empty record type in negative position cannot be obtained from the instantiation of a type variable in negative position, constrained by a record type, because the $UB_\top(\ldots,\ldots)$ function that computes instantiations inserts the l field. The only other possibility is that an empty record type is obtained from instantiating β, occurring in positive position; but then an argument similar to before demonstrates that $\theta(\alpha)$ and $\theta(\beta)$ cannot have a common path to an empty record type, because such a type in the range of $\theta(\alpha)$ must arise from an instantiation containing $\theta(\beta)$ embedded in the arguments to a type constructor. □

Theorem 48 (Decision procedure for semantic entailment). *To decide $\sigma\leqslant^{\forall}_{sem}\sigma'$, given $\sigma=\forall\overline{\alpha_m}\leqslant\overline{\tau_m}.\tau$ and $\sigma'=\forall\overline{\beta_n}\leqslant\overline{\tau'_n}.\tau'$. Define*

$$\mathcal{Q}=\forall\overline{\beta_n}\leqslant\overline{\tau'_n}.\exists\overline{\alpha_m},$$
$$C=\{\overline{\alpha_m}\leqslant\overline{\tau_m}\}\cup\{\tau\leqslant\tau'\}.$$

Then $\sigma\leqslant^{\forall}_{sem}\sigma'$ if and only if $\models\mathcal{Q}C$.

Proof. Define $\Gamma=\{\overline{\beta_n}\leqslant\overline{\tau'_n}\}$. For the "if" part, if $\theta\models\mathcal{Q}C$ for some θ, then $\Gamma\vdash\tau_1\leqslant\tau_2$ for all $(\tau_1\leqslant\tau_2)\in\theta(C)$. Let τ'' be an instance of σ', so $\tau''=\theta'(\tau')$ for some θ' such that $\theta'\models\Gamma$. By soundness of the subtype logic, $\Gamma\models\overline{\theta(\alpha_m)}\leqslant\overline{\theta(\tau_m)}$ and $\Gamma\models\theta(\tau)\leqslant\theta(\tau')$, so that $\vdash\overline{\theta'\theta(\alpha_m)}\leqslant\overline{\theta'\theta(\tau_m)}$ and $\Gamma\models\theta'\theta(\tau)\leqslant\theta'\theta(\tau')$. Furthermore $\theta'\theta(\tau')=\theta(\tau')$. So $\theta'\theta(\tau)$ is a valid instance of σ, and $\vdash\theta'\theta(\tau)\leqslant\theta(\tau')$.

For the "only if" part, assume $\sigma\leqslant^{\forall}_{sem}\sigma'$. If τ'' is an instance of σ', then $\tau''=\theta(\tau')$ for some θ such that $\theta\models\Gamma$. So $\sigma\leqslant^{\forall}_{sem}\sigma'$ is equivalent to $\Gamma\models\theta(\tau)\leqslant\theta(\tau')$ for some θ such that $\Gamma\models\overline{\theta(\alpha_m)}\leqslant\overline{\theta(\tau_m)}$. By completeness of the subtype logic, this is equivalent to $\Gamma\vdash\theta(\tau)\leqslant\theta(\tau')$ for some θ such that $\Gamma\vdash\overline{\theta(\alpha_m)}\leqslant\overline{\theta(\tau_m)}$. By the definition of derivability in the subtype logic, this is equivalent to $\models\forall\overline{\beta_n}\leqslant\overline{\tau'_n}.\exists\overline{\alpha_m}.C$. □

9. Conclusions

Finite subtype inference occupies a middle ground between Hindley–Milner-type inference (as in ML) and subtype inference with recursively constrained types. We have

presented algorithms for finite subtype inference, including checking for entailment of inferred types against explicitly declared polymorphic types. This resolves for finite types a problem that is still open for recursively constrained types. Some motivation for this work, particularly for finite types and explicit polymorphism, is in providing subtype inference for first-class container objects with polymorphic methods.

Flanagan and Felleisen give algorithms for checking entailment of recursive set constraints arising from data-flow analysis [8]. Although superficially similar to constraint systems for object-oriented languages, there are, in fact, subtle but significant differences in the underlying languages, and it is not clear if their techniques can be adapted to solve the problem of entailment for recursive types.

Bourdoncle and Merz [4] provide a type-checking algorithm for an ML dialect where subtyping is declared between class types. Their work is more related to earlier work on finite atomic subtyping than the current work, and they do not have a structural notion of subtyping.

Sequeira [26] investigates type inference for a language with bounded polymorphic types, extending some of the approach of Odersky and Läufer [19] to a language with subtyping. As with the current work, he considers a type system without $\bot$ and $\top$, and to this purpose applies the theory of Helly posets to investigate the problems of entailment and solvability. Although he gives an algorithm for deciding entailment, this algorithm is for a type system without record types. Kaes [13] and Smith [27] provide algorithms for checking solvability of subtyping and overloading constraints with finite types, while Smith considers the problem of entailment in his system. However, again these approaches are only concerned with structural subtyping, nor do they consider scoping of type variables.

The other relevant work is that of Pierce and Turner [22] on local type inference. They work with an impredicative-type system, and allow type annotations to be omitted where it is possible to infer the type from an "upper bound" type constraint. The work presented here may be seen as a more flexible approach, where type annotations are only required for polymorphic function definitions. Pierce and Turner require that it be possible (using meets and joins) to compute the specific type of every expression. Where a type variable occurs in invariant position (for example, in the element type of a mutable reference cell), their algorithm may fail to determine a type. Pottier reports that type variables in invariant position are reasonably common in practical subtype inference. Although we have not elaborated upon it here, impredicativity can be incorporated using objects with polymorphic methods [5].

Acknowledgements

Thanks to Fritz Henglein, Benjamin Pierce and Scott Smith for helpful discussions. Thanks to the anonymous reviewers for their careful comments that greatly improved the content and the presentation of the paper.

References

[1] M. Abadi, L. Cardelli, A Theory of Objects, Springer, Berlin, 1996.
[2] A. Aiken, E. Wimmers, Solving systems of set constraints, Proc. IEEE Symp. on Logic in Computer Science, Santa Cruz, CA, June 1992, pp. 329–340.
[3] A. Aiken, E. Wimmers, Type inclusion constraints and type inference, Proc. ACM Symp. on Functional Programming and Computer Architecture, Copenhagen, Denmark, June 1993, pp. 31–41.
[4] F. Bourdoncle, S. Merz, Type checking higher-order polymorphic multi-methods, Proc. ACM Symp. on Principles of Programming Languages, Paris, France, 1997.
[5] D. Duggan, Object type constructors, submitted for publication, A preliminary version appeared in the Workshop on Foundations of Object-Oriented Languages, New Brunswick, NJ, July 1996.
[6] D. Duggan, Unification with extended patterns, Theoret. Comput. Sci. 206 (1998) 1–50.
[7] J. Eifrig, S. Smith, V. Trifonov, Type inference for recursively constrained types and its application to oop, Proc. Conf. on Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science, Springer, Berlin, 1995, to appear.
[8] C. Flanagan, M. Felleisen, Componential set-based analysis, Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Las Vegas, NV, 1997.
[9] You-Chin Fuh, P. Mishra, Type inference with subtypes, Theoret. Comput. Sci. 73 (1990) 155–175.
[10] F. Henglein, J. Rehof, Constraint automata and the complexity of recursive subtype entailment, in: ICALP: Ann. Internat. Colloq. on Automata, Languages and Programming, Aalborg, Denmark, 1998.
[11] F. Henglein, J. Rehof, The complexity of subtype entailment for simple types, Proc. 12 Ann. IEEE Symp. on Logic in Computer Science, IEEE Computer Soc. Press, Silver Spring, MD, Warsaw, Poland, 29 June–2 July 1997, pp. 352–361.
[12] M. Jones, First-class polymorphism with type inference, Proc. ACM Symp. on Principles of Programming Languages, January 1997, ACM Press, New York.
[13] S. Kaes, Type inference in the presence of overloading, subtyping and recursive types, Proc. ACM Symp. on Lisp and Functional Programming, 1992, ACM Press, New York, pp. 193–204.
[14] X. Leroy, M. Mauny, Dynamics in ML, J. Functional Programming 3 (4) (1993) 431–463.
[15] P. Lincoln, J.C. Mitchell, Algorithmic aspects of type inference with subtypes, Proc. ACM Symp. on Principles of Programming Languages, 1992, ACM Press, New York, pp. 293–304.
[16] D. Miller, Unification under a mixed prefix, J. Symbolic Comput. 14 (1992) 321–358.
[17] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (1978) 343–378.
[18] J.C. Mitchell, Type inference with simple subtypes, J. Funct. Programming 1 (3) (1991) 245–286.
[19] M. Odersky, K. Läufer, Putting type annotations to work, Proc. ACM Symp. on Principles of Programming Languages, St. Petersburg, FL, 1996.
[20] J. Palsberg, Efficient inference of object types, Inform. and Comput. 123 (2) (1995) 198–209.
[21] J. Palsberg, M. Wand, P. O'Keefe, Type inference with non-structural subtyping, Formal Aspects Comput. 9 (1997) 49–67.
[22] B. Pierce, D. Turner, Local type inference, Proc. ACM Symp. on Principles of Programming Languages, San Diego, CA, ACM Press, New York, January 1998.
[23] F. Pottier, Simplifying subtyping constraints, Proc. ACM Internat. Conf. on Functional Programming, Philadelphia, PA, ACM Press, New York, May 1996.
[24] D. Rémy, Programming objects with ML-ART: an extension to ml with abstract and record types, in: M. Hagiya, J.C. Mitchell (Eds.), Internat. Symp. on Theoretical Aspects of Computer Software, Sendai, Japan, Springer, Berlin, April 1994, pp. 321–346.
[25] G. Sénizergues, The equivalence problem for deterministic pushdown automata is decidable, Proc. Internat. Conf. on Automata, Languages and Programming, Lecture Notes in Computer Science, Springer, Berlin, 1997 pp. 671–681.
[26] D. Sequeira, Type inference with bounded quantification. Ph.D. Thesis, University of Edinburgh, 1998.
[27] G. Smith, Principal type schemes for functional programs with overloading and subtyping, Sci. Comput. Programming 23 (2/3) (1994) 197–226.
[28] M. Solomon, Type definitions with parameters, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1978, pp. 31–38.
[29] V. Trifonov, S. Smith, Subtyping constrained types, Static Analysis Symp. Lecture Notes in Computer Science, Vol. 1145, Springer, Berlin, 1996, pp. 349–365.