Subtyping Existential Types

Stefan Wehr and Peter Thiemann

Institut für Informatik, Universität Freiburg
{wehr,thiemann}@informatik.uni-freiburg.de

June 9, 2008
Technical Report No. 240

Abstract Constrained existential types are a powerful language feature that subsumes Java-like interface and wildcard types. But existentials do not mingle well with subtyping: subtyping is already undecidable for very restrictive settings. This paper defines two subtyping relations by extracting the features specific to existentials from current language proposals (JavaGI, WildFJ, Scala). Both subtyping relations are undecidable. The paper also discusses the consequences of removing existentials from JavaGI and possible amendments to regain their features.

1 Introduction

Constrained existential types (also called "bounded existential types" [5, 12]) arise from the need for structured and partial data abstraction and information hiding. They have found uses for modeling object oriented languages in general [2], as well as for modeling specific features such as Java wildcards [3,4,18,19] and Java-like interface types in the JavaGI language [21]. In fact, JavaGI supports general existential types and provides interface types as a special case supported by syntactic sugar. Building directly on existential types has several advantages compared to interface types: they allow the general composition of interface types, they encompass Java wildcards, and they enable meaningful types in the presence of multi-headed interfaces.[1]

Work on the type checker for an implementation of JavaGI uncovered the consequences of supporting general existential types. They do not only introduce a wealth of complexity into the type system (something we can live with) but they may also cause nontermination in the type checker (something we cannot live with): JavaGI's subtyping relation with existential types is undecidable.

After establishing some background on JavaGI (§ 2), we define two calculi with constrained existential types and subtyping. The first calculus (§ 3) is a subset of JavaGI's formalization [21]. The second calculus (§ 4) supports existential types with lower and upper bounds, very much like Scala [13] and formal systems for modeling Java wildcards [3, 4, 18]. We prove that the subtyping relations of both calculi are undecidable.

Furthermore, we discuss alternative design options for JavaGI that avoid the use of general existential types but keep the remaining features (§ 5). Finally, we review related work (§ 6) and conclude (§ 7). Detailed proofs can be found in the appendices.

[1] JavaGI provides multi-headed interfaces that abstract over a family of types.

2 Background

JavaGI [21] is a conservative extension of Java 1.5 that generalizes Java's interface concept to incorporate the essential features of Haskell type classes [8,9,20]. This generalization allows for retroactive and constrained interface implementations, binary methods, static methods in interfaces, default implementations for interface methods, and multi-headed interfaces (interfaces over families of types). Furthermore, JavaGI generalizes Java-like interface types to existential types. This section only discusses the features relevant to this paper, namely retroactive interface implementations and existential types, and ignores the rest.

2.1 Retroactive Interface Implementations

A class definition in Java must specify all interfaces that the class implements. In contrast, JavaGI enables programmers to add implementations for interfaces to existing classes at any time. For example, Java rejects the use of a **for**-loop to iterate over the characters of a string because the class `String` does not implement the interface `Iterable`:[2]

```
for (Character c : someString) { ... } // illegal in Java
```

As the definition of class `String` is fixed, there is no hope of getting this code working. In contrast, JavaGI allows the retroactive implementation of `Iterable`:[3]

```
implementation Iterable<Character> [String] {
  public Iterator<Character> iterator() {
    return new Iterator<Character>() {
      private int index = 0;
      public boolean hasNext() { return index < length(); }
      public Character next() { return charAt(index++); }
    };
  }
}
```

This *implementation definition* specifies that the *implementing type* `String`, enclosed in square brackets `[]`, implements the interface `Iterable<Character>`. The definition of the `iterator` method can use the methods `length` and `charAt` because they are part of `String`'s public interface.

2.2 Constrained Existential Types

Java uses the name of an interface as an *interface type* that denotes the set of all types implementing the interface. Instead of interface types, JavaGI features *constrained existential types* (*existentials* for short) and provides syntactic sugar for recovering interface types. For example, the interface type `List<String>` abbreviates the existential type ∃ `X` **`where`** `X` **`implements`** `List<String> . X`. The *implementation constraint* "`X` **`implements`** `List<String>`" restricts instantiations

[2] Java's enhanced **for**-loop allows to iterate over arrays and all types implementing the `Iterable<X>` interface, which contains a single method `Iterator<X> iterator()`.

[3] We ignore the `remove` method of the `Iterator` interface.

of the type variable `X` to types that implement the interface `List<String>`. Thus, the existential type denotes the set of all types implementing `List<String>`, exactly like the synonymous interface type.

Existentials are more general than interface types. For instance, the existential `∃ X where X implements List<String>, X implements Set<String> . X` denotes the set of all types that implement both `List<String>` and `Set<String>`. Java supports such intersections of interface types only for specifying bounds of type variables. Existentials also encompass Java wildcards [3, 4, 18, 19]. For instance, the existential type `∃ X where X extends Number . List<X>` corresponds to the wildcard type `List<? extends Number>`.[4]

JavaGI allows implementation definitions for existentials. For example, a programmer may write an implementation definition to specify that all types implementing `List<X>` also implement `Iterable<X>`. Such a definition is feasible because iterators can be implemented using only operations of the `List` interface. The example also demonstrates that JavaGI supports *generic implementation definitions*, which are parameterized by type variables.

```
implementation<X> Iterable<X> [∃ L where L implements List<X> . L] {
  Iterator<X> iterator() { return new Iterator<X>() {
      /* as for String, replacing length with size and charAt with get. */ };
  }
}
```

A Java programmer would have to implement `Iterable` from scratch for every class that implements `List`. Abstract classes do not help with this problem because Java does not support multiple inheritance.

3 Subtyping Existential Types with Implementation Constraints

This section introduces $\mathcal{EX}_{impl}$, a subtyping calculus with existentials and implementation constraints. $\mathcal{EX}_{impl}$ is a subset of Core–JavaGI from the original formulation of JavaGI's type system [21]. It does not model all aspects of JavaGI, but contains only those features that make subtyping undecidable.

3.1 Definition of $\mathcal{EX}_{impl}$

Fig. 1 defines the syntax, as well as the entailment and subtyping relations of $\mathcal{EX}_{impl}$. A type T is either a type variable X or an existential $\exists X\,\mathtt{where}\,\overline{P}\,.\,X$. For simplicity, there are no class types, existentials have a single quantified type variable, and the body of an existential must be the quantified type variable.[5] Overbar notation $\overline{\xi}$ denotes a sequence $\xi_1, \ldots, \xi_n$ of syntactic entities with $\bullet$ standing for the empty sequence. Sometime, the sequence $\overline{\xi}$ stands for the set $\{\overline{\xi}\}$. Existentials are considered equal up to renaming of bound type variables, reordering of constraints, and elimination of duplicate constraints.

[4] Because `List` is an interface, `∃ X where X extends Number . List<X>` stands for `∃ X,L where X extends Number, L implements List<X> . L`

[5] The body of an existential is the part after the ".".

$$\begin{aligned} T,U,V,W &::= X \mid \exists X\,\texttt{where}\,\overline{P}\,.\,X \\ P,Q,R &::= X\,\texttt{implements}\,I\texttt{<}\overline{T}\texttt{>} \\ \mathit{def} &::= \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>} \mid \texttt{implementation<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ \texttt{[}T\texttt{]} \end{aligned}$$

$$\text{E}_1\text{-IMPL}\quad \frac{\texttt{implementation<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ \texttt{[}U\texttt{]} \in \Theta}{\Theta;\Delta \Vdash [\overline{V/X}]\big(U\,\texttt{implements}\,I\texttt{<}\overline{T}\texttt{>}\big)} \qquad \text{E}_1\text{-LOCAL}\quad \frac{P \in \Delta}{\Theta;\Delta \Vdash P} \qquad \text{S}_1\text{-REFL}\quad \Theta;\Delta \vdash T \leq T$$

$$\text{S}_1\text{-TRANS}\quad \frac{\Theta;\Delta \vdash T \leq U \qquad \Theta;\Delta \vdash U \leq V}{\Theta;\Delta \vdash T \leq V} \qquad \text{S}_1\text{-OPEN}\quad \frac{\Theta;\Delta,\overline{P} \vdash X \leq T \qquad X \notin \mathsf{ftv}(\Theta,\Delta,T)}{\Theta;\Delta \vdash \exists X\,\texttt{where}\,\overline{P}\,.\,X \leq T}$$

$$\text{S}_1\text{-ABSTRACT}\quad \frac{(\forall i)\ \Theta;\Delta \Vdash [T/X]P_i}{\Theta;\Delta \vdash T \leq \exists X\,\texttt{where}\,\overline{P}\,.\,X}$$

Fig. 1. Type syntax, entailment, and subtyping for $\mathcal{EX}_{impl}$.

An implementation constraint P has the form $X\,\texttt{implements}\,I\texttt{<}\overline{T}\texttt{>}$ and constrains the type variable X to types that implement the interface $I\texttt{<}\overline{T}\texttt{>}$. In comparison with upper bounds for type variables, implementation constraints allow more precise typings, especially for binary methods [1]. An interface without type parameters is written I instead of $I\texttt{<}\bullet\texttt{>}$.

A definition *def* in $\mathcal{EX}_{impl}$ is either an interface or an implementation definition. Interface and implementation definitions do not have method signatures or bodies, because they do not matter for the entailment and subtyping relation of $\mathcal{EX}_{impl}$. Moreover, $\mathcal{EX}_{impl}$ does not support interface inheritance. A program environment Θ is a finite set of definitions *def*, and a type environment Δ a finite set of constraints P, where Δ, P abbreviates $\Delta \cup \{P\}$.

The entailment relation $\Theta;\Delta \Vdash T\,\texttt{implements}\,I\texttt{<}\overline{T}\texttt{>}$ expresses that type T implements interface $I\texttt{<}\overline{T}\texttt{>}$. A type implements an interface either because it corresponds to an instance of a suitable implementation definition (rule E$_1$-IMPL) or because the type environment contains the constraint (rule E$_1$-LOCAL). The notation $[\overline{T/X}]$ stands for the capture-avoiding substitution replacing each X_i with T_i. Full JavaGI uses the entailment relation (among other things) to verify that the instantiation of a generic class or method fulfills the implementation constraints associated with that class or method.

The subtyping relation $\Theta;\Delta \vdash T \leq U$ states that T is a subtype of U. It is reflexive and transitive as usual. Rule S$_1$-OPEN opens an existential on the left-hand side of a subtyping judgment by moving its constraints into the type environment. The premise $X \notin \mathsf{ftv}(\Theta,\Delta,T)$ ensures that the existentially quantified type variable is sufficiently fresh and does not escape from its scope. Rule S$_1$-ABSTRACT deals with existentials on the right-hand side of a subtyping judgment. It states that T is a subtype of some existential if all constraints of the existential hold after substituting T for the existentially quantified type variable.

While developing a type soundness proof for Core–JavaGI, we verified that the subtyping relation of $\mathcal{EX}_{impl}$ supports the usual principle of subsumption: we can always promote the type of an expression to some supertype without causing runtime errors.

3.2 Undecidability of Subtyping in $\mathcal{EX}_{impl}$

We prove undecidability of subtyping in $\mathcal{EX}_{impl}$ by reduction from Post's Correspondence Problem (PCP). It is well known that PCP is undecidable [7, 17].

Definition 1 (PCP). *Let $\{(u_1, v_1) \dots, (u_n, v_n)\}$ be a set of pairs of non-empty words over some finite alphabet Σ with at least two elements. A solution of PCP is a sequence of indices $i_1 \dots i_r$ such that $u_{i_1} \dots u_{i_r} = v_{i_1} \dots v_{i_r}$. The decision problem asks whether such a solution exists.*

Theorem 1. *Subtyping in $\mathcal{EX}_{impl}$ is undecidable.*

Proof. Let $\mathcal{P} = \{(u_1, v_1), \dots, (u_n, v_n)\}$ be a particular instance of PCP over the alphabet Σ. We can encode $\mathcal{P}$ as an equivalent subtyping problem in $\mathcal{EX}_{impl}$ as follows. First, words over Σ must be represented as types in $\mathcal{EX}_{impl}$.

```
interface E       // empty word ε
interface L<X>    // letter, for every L ∈ Σ
```

Words $u \in \Sigma^*$ are formed with these interfaces through nested existentials. For example, the word AB is represented by

$$\exists X\ \texttt{where}\ X\ \texttt{implements}\ \texttt{A<}\exists Y\ \texttt{where}\ Y\ \texttt{implements}\ \texttt{B<}\\ \exists Z\ \texttt{where}\ Z\ \texttt{implements}\ \texttt{E}\,.\,Z\texttt{>}\,.\,Y\texttt{>}\,.\,X$$

The abbreviation $\exists I\texttt{<}\overline{T}\texttt{>}$ stands for the type $\exists X\ \texttt{where}\ X\ \texttt{implements}\ I\texttt{<}\overline{T}\texttt{>}\,.\,X$. Using this notation, the word AB is represented by $\exists\texttt{A<}\exists\texttt{B<}\exists\texttt{E>>}$.

Formally, we define the representation of a word u as $[\![u]\!] = u \# \exists\texttt{E}$, where $u \# T$ is the concatenation of a word u with a type T:

$$\varepsilon \# T \triangleq T \qquad\qquad Lu \# T \triangleq \exists L\texttt{<}u \# T\texttt{>}$$

Two interfaces are required to model the search for a solution of PCP:

```
interface S<X,Y>    // search state
interface G         // search goal
```

The type $\exists\texttt{S<}[\![u]\!], [\![v]\!]\texttt{>}$ represents a particular search state where we have already accumulated indices $i_1, \dots, i_k$ such that $u = u_{i_1} \dots u_{i_k}$ and $v = v_{i_1} \dots v_{i_k}$. To model valid transitions between search states, we define implementations of S for all $i \in \{1, \dots, n\}$ as follows:

$$\texttt{implementation<X,Y> S<}u_i\texttt{\#X, }v_i\texttt{\#Y> [}\exists\texttt{ S<X,Y>]} \tag{1}$$

The type $\exists\texttt{G}$ represents the goal of a search, as expressed by the following implementation:

$$\texttt{implementation<X> G [}\exists\texttt{ S<X,X>]} \tag{2}$$

To get the search running we ask whether there exists some $i \in \{1, \dots, n\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash \exists\texttt{S<}[\![u_i]\!], [\![v_i]\!]\texttt{>} \leq \exists\texttt{G}$ is derivable. The program $\Theta_{\mathcal{P}}$ consists of the interfaces and implementations just defined. In Appendix A, we prove that the given PCP instance $\mathcal{P}$ has a solution if and only if there exists some $i \in \{1, \dots, n\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash \exists\texttt{S<}[\![u_i]\!], [\![v_i]\!]\texttt{>} \leq \exists\texttt{G}$ is derivable. □

$$\begin{aligned} N, M &::= C\texttt{<}\overline{X}\texttt{>} \mid \texttt{Object} \\ T, U, V, W &::= X \mid N \mid \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N \\ P, Q, R &::= X\,\texttt{extends}\,T \mid X\,\texttt{super}\,T \end{aligned}$$

$$\textsc{E2-extends}\ \frac{\Delta \vdash T \leq U}{\Delta \Vdash T\,\texttt{extends}\,U} \qquad \textsc{E2-super}\ \frac{\Delta \vdash U \leq T}{\Delta \Vdash T\,\texttt{super}\,U} \qquad \textsc{S2-refl}\ \Delta \vdash T \leq T \qquad \textsc{S2-trans}\ \frac{\Delta \vdash T \leq U \qquad \Delta \vdash U \leq V}{\Delta \vdash T \leq V}$$

$$\textsc{S2-object}\ \Delta \vdash T \leq \texttt{Object} \qquad \textsc{S2-extends}\ \frac{X\,\texttt{extends}\,T \in \Delta}{\Delta \vdash X \leq T} \qquad \textsc{S2-super}\ \frac{X\,\texttt{super}\,T \in \Delta}{\Delta \vdash T \leq X}$$

$$\textsc{S2-open}\ \frac{\Delta, \overline{P} \vdash N \leq T \qquad \overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset}{\Delta \vdash \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N \leq T} \qquad \textsc{S2-abstract}\ \frac{T = [\overline{U/X}]N \qquad (\forall i)\ \Delta \Vdash [\overline{U/X}]P_i}{\Delta \vdash T \leq \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N}$$

Fig. 2. Syntax, Entailment, and Subtyping for $\mathcal{EX}_{uplo}$

4 Subtyping Existential Types with Upper and Lower Bounds

This section considers the calculus $\mathcal{EX}_{uplo}$, which is similar in spirit to $\mathcal{EX}_{impl}$, but supports upper and lower bounds for type variables and no implementation constraints. Other researchers [3,4,18] use formal systems very similar to $\mathcal{EX}_{uplo}$ for modeling Java wildcards [19]. It is not the intention of $\mathcal{EX}_{uplo}$ to provide another formalization of wildcards, but rather to expose the essential ingredients that make subtyping undecidable in a calculus as simple as possible.

4.1 Definition of $\mathcal{EX}_{uplo}$

Fig. 2 defines the syntax and the entailment and subtyping relations of $\mathcal{EX}_{uplo}$. A class type N is either `Object` or an instantiated generic class $C\texttt{<}\overline{X}\texttt{>}$, where the type arguments must be type variables. A type T is a type variable, a class type, or an existential. Unlike in $\mathcal{EX}_{impl}$, existentials in $\mathcal{EX}_{uplo}$ may quantify over several type variables and the body of an existential must be a class type. A constraint P places either an upper bound ($X\,\texttt{extends}\,T$) or a lower bound ($X\,\texttt{super}\,T$) on a type variable X. Type environments Δ are defined as for $\mathcal{EX}_{impl}$.

Class definitions and inheritance are omitted from $\mathcal{EX}_{uplo}$. The only assumption is that every class name C comes with a fixed arity that is respected when applying C to type arguments. There are some further restrictions:

(1) If $T = \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N$, then $\overline{X} \neq \bullet$ and $\overline{X} \subseteq \mathsf{ftv}(N)$.
(2) If $T = \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N$ and $P \in \overline{P}$, then $P = Y\,\texttt{extends}\,T$ or $P = Y\,\texttt{super}\,T$ with $Y \in \overline{X}$. That is, only bound variables may be constrained.
(3) A type variable must not have both upper and lower bounds.[6]

Constraint entailment ($\Delta \Vdash T\,\texttt{extends}\,U$ and $\Delta \Vdash U\,\texttt{super}\,T$) uses subtyping ($\Delta \vdash T \leq U$) to check that the constraint given holds. The subtyping rules for $\mathcal{EX}_{uplo}$ are similar to those for $\mathcal{EX}_{impl}$, except that `Object` is now a supertype of every type and that rules S2-EXTENDS and S2-SUPER use assumptions from Δ.

[6] Modeling Java wildcards requires upper and lower bounds for the same type variable in certain situations.

$$
\begin{array}{rcll}
\tau^+ & ::= & \mathsf{Top} \mid \forall \alpha_0 {\le} \tau_0^- \ldots \alpha_n {\le} \tau_n^- \,.\, \neg\, \tau^- & (n \in \mathbb{N}) \\
\tau^- & ::= & \alpha \mid \forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \tau^+ & (n \in \mathbb{N}) \\
\Gamma^- & ::= & \emptyset \mid \Gamma^-, \alpha {\le} \tau^- &
\end{array}
$$

$$
\textsc{d-top}\quad \Gamma \vdash \tau \le \mathsf{Top}
\qquad
\textsc{d-var}\quad \frac{\tau \neq \mathsf{Top} \qquad \Gamma \vdash \Gamma(\alpha) \le \tau}{\Gamma \vdash \alpha \le \tau}
\qquad
\textsc{d-all-neg}\quad \frac{\Gamma, \alpha_0 {\le} \phi_0 \ldots \alpha_n {\le} \phi_n \vdash \tau \le \sigma}{\Gamma \vdash \forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma \le \forall \alpha_0 {\le} \phi_0 \ldots \alpha_n {\le} \phi_n \,.\, \neg\, \tau}
$$

Fig. 3. Syntax and Subtyping for $F_{\le}^{D}$

4.2 Undecidability of Subtyping in $\mathcal{EX}_{uplo}$

The undecidability proof of subtyping in $\mathcal{EX}_{uplo}$ is by reduction from $F_{\le}^{D}$ [14], a restricted version of $F_{\le}$ [5]. Pierce defines $F_{\le}^{D}$ for his undecidability proof of $F_{\le}$ subtyping [14].

Fig. 3 defines the syntax and the subtyping relation of $F_{\le}^{D}$. A Type τ is either a n-positive type, τ^+, or a n-negative type, τ^-, where n is a fixed natural number standing for the number of type variables (minus one) bound at the top-level of the type. A n-negative type environment Γ^- associates type variables α with upper bounds τ^-. The polarity ($+$ or $-$) characterizes at which positions of a subtyping judgment a type or type environment may appear. For readability, we often omit the polarity and leave n implicit.

A *n-ary subtyping judgment* in $F_{\le}^{D}$ has the form $\Gamma^- \vdash \sigma^- \le \tau^+$, where Γ^- is a n-negative type environment, σ^- is a n-negative type, and τ^+ is a n-positive type. Only n-negative types appear to the left and only n-positive types appear to the right of the $\le$ symbol. The subtyping rule D-ALL-NEG compares two quantified types $\sigma = \forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma'$ and $\tau = \forall \alpha_0 {\le} \tau_0 \ldots \alpha_n {\le} \alpha_n \,.\, \neg\, \tau'$ by swapping the left- and right-hand sides of the subtyping judgment and checking $\tau' \le \sigma'$ under the extended environment $\Gamma, \alpha_0 {\le} \tau_0 \ldots \alpha_n {\le} \tau_n$. The rule is correct with respect to $F_{\le}$ because we may interpret every $F_{\le}^{D}$ type as an $F_{\le}$ type:

$$
\begin{array}{rl}
\forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma' = \forall \alpha_0 {\le} \mathsf{Top} \ldots \forall \alpha_n {\le} \mathsf{Top}. \forall \beta {\le} \sigma' \,.\, \beta & (\beta \text{ fresh}) \\
\forall \alpha_0 {\le} \tau_0 \ldots \alpha_n {\le} \alpha_n \,.\, \neg\, \tau' = \forall \alpha_0 {\le} \tau_0 \ldots \forall \alpha_n {\le} \tau_n . \forall \beta {\le} \tau' \,.\, \beta & (\beta \text{ fresh})
\end{array}
$$

Using these abbreviations, every $F_{\le}^{D}$ subtyping judgment can be read as an $F_{\le}$ subtyping judgment. The subtype relations in $F_{\le}^{D}$ and $F_{\le}$ coincide for judgments in their common domain [14].

It is sufficient to consider only *closed judgments*. A type τ is closed under Γ if $\mathsf{ftv}(\tau) \subseteq \mathsf{dom}(\Gamma)$ (where $\mathsf{dom}(\alpha_1 {\le} \tau_1, \ldots, \alpha_n {\le} \tau_n) = \{\alpha_1, \ldots, \alpha_n\}$) and, if $\tau = \forall \alpha_0 {\le} \tau_0 \ldots \alpha_n {\le} \tau_n \,.\, \neg\, \sigma$, then no α_i appears free in any τ_j. A type environment Γ is closed if $\Gamma = \emptyset$ or $\Gamma = \Gamma', \alpha {\le} \tau$ with Γ' closed and τ closed under Γ'. A judgment $\Gamma \vdash \tau \le \sigma$ is closed if Γ is closed and τ, σ are closed under Γ.

We now come to the central theorem of this section.

Theorem 2. *Subtyping in $\mathcal{EX}_{uplo}$ is undecidable.*

Proof. The proof is by reduction from $F_{\le}^{D}$. Fig. 4 defines a translation from $F_{\le}^{D}$ types, type environments, and subtyping judgments to their corresponding $\mathcal{EX}_{uplo}$ forms. The translation of an n-ary subtyping judgment assumes the

$$
\begin{aligned}
\llbracket \mathsf{Top} \rrbracket^{+} &= \texttt{Object} \\
\llbracket \forall \alpha_0 {\le} \tau_0^{-} \ldots \alpha_n {\le} \tau_n^{-} \,.\, \neg\, \tau^{-} \rrbracket^{+} &= \neg\, \exists Y, \overline{X^{\alpha_i}} \;\texttt{where}\; X^{\alpha_0} \;\texttt{extends}\; \llbracket \tau_0 \rrbracket^{-} \ldots \\
&\qquad\qquad X^{\alpha_n} \;\texttt{extends}\; \llbracket \tau_n \rrbracket^{-}, Y \;\texttt{extends}\; \llbracket \tau \rrbracket^{-} \,.\, C^{n+2}\texttt{<}Y, \overline{X^{\alpha_i}}\texttt{>} \\
\llbracket \alpha \rrbracket^{-} &= X^{\alpha} \\
\llbracket \forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \tau^{+} \rrbracket^{-} &= \neg\, \exists Y, \overline{X^{\alpha_i}} \;\texttt{where}\; Y \;\texttt{extends}\; \llbracket \tau \rrbracket^{+} \,.\, C^{n+2}\texttt{<}Y, \overline{X^{\alpha_i}}\texttt{>} \\
\llbracket \emptyset \rrbracket^{-} &= \emptyset \\
\llbracket \Gamma, \alpha {\le} \tau^{-} \rrbracket^{-} &= \llbracket \Gamma \rrbracket^{-}, X^{\alpha} \;\texttt{extends}\; \llbracket \tau \rrbracket^{-} \\
\llbracket \Gamma^{-} \vdash \tau^{-} \le \sigma^{+} \rrbracket &= \llbracket \Gamma \rrbracket^{-} \vdash \llbracket \tau \rrbracket^{-} \le \llbracket \sigma \rrbracket^{+} \\
\neg\, T &\equiv \exists X \;\texttt{where}\; X \;\texttt{super}\; T \,.\, D^{1}\texttt{<}X\texttt{>}
\end{aligned}
$$

Fig. 4. Reduction from $F_{\le}^{D}$ to $\mathcal{EX}_{uplo}$

existence of two $\mathcal{EX}_{uplo}$ classes: C^{n+2} accepts $n+2$ type arguments, and D^1 takes one type argument. The superscripts in $\llbracket \cdot \rrbracket^{+}$ and $\llbracket \cdot \rrbracket^{-}$ indicate whether the translation acts on positive or negative entities.

An n-positive type $\forall \alpha_0 {\le} \tau_0^{-} \ldots \alpha_n {\le} \tau_n^{-} \,.\, \neg\, \tau^{-}$ is translated into an negated existential. The existentially quantified type variables $X^{\alpha_0}, \ldots, X^{\alpha_n}$ correspond to the universally quantified type variables $\alpha_0, \ldots, \alpha_n$. The bound $\llbracket \tau \rrbracket^{-}$ of the fresh type variable Y represents the body $\neg\, \tau^{-}$ of the original type. We cannot use $\llbracket \tau \rrbracket^{-}$ directly as the body because existentials in $\mathcal{EX}_{uplo}$ have only class types as their bodies. The translation for n-negative types is similar to the one for n-positive types. It is easy to see that the $\mathcal{EX}_{uplo}$ types in the image of the translation meet the restrictions defined in Section 4.1. Type environments and subtyping judgments are translated in the obvious way.

A negated type, written $\neg\, T$, is an abbreviation for an existential with a single `super` constraint: $\neg\, T \equiv \exists X \;\texttt{where}\; X \;\texttt{super}\; T \,.\, D^{1}\texttt{<}X\texttt{>}$, where X is fresh. The `super` constraint simulates the behavior of the $F_{\le}^{D}$ subtyping rule D-ALL-NEG, which swaps the left- and right-hand sides of subtyping judgments.

We now need to verify that $\Gamma \vdash \tau \le \sigma$ is derivable in $F_{\le}^{D}$ if and only if $\llbracket \Gamma \vdash \tau \le \sigma \rrbracket$ is derivable in $\mathcal{EX}_{uplo}$. The "⇒" direction is an easy induction on the derivation of $\Gamma \vdash \tau \le \sigma$. The "⇐" direction requires more work because the transitivity rule S$_2$-TRANS (Fig. 2) involves an intermediate type which is not necessarily in the image of the translation. Hence, a direct proof by induction on the derivation of $\llbracket \Gamma \vdash \tau \le \sigma \rrbracket$ fails. To solve this problem, we give an equivalent definition of the $\mathcal{EX}_{uplo}$ subtyping relation that does not include an explicit transitivity rule. See Appendix B for details and the full proofs. □

5 Lessons Learned

What are the consequences of this investigation for the design of JavaGI? While existentials are powerful and unify several diverse concepts, they complicate the metatheory of JavaGI considerably. Also, subtyping with existentials is undecidable even under severe restrictions.

The initial development of JavaGI's metatheory uses existentials and imposes several restrictions to ensure decidability of subtyping. However, these restrictions are difficult to explain to users of JavaGI because they seem ad-hoc. Our present view is that existentials may not be worth all the trouble. After all, JavaGI's main feature is its very general and powerful interface concept (which this paper does not explore). Hence, the upcoming revision of JavaGI's design has all features of the original design but it does not require existentials in their full generality. It gives up some of the power in favor of simplicity. Several other features make up for the lack of existentials and experience will show whether this design is satisfactory.

In fact, the upcoming revision of JavaGI copes with all the uses of existentials in JavaGI [21] as mentioned in the introduction.

General composition of interface types. The revised design supports Java-like interface types and intersections thereof.

Wildcards. The revised design does not encode wildcards through existentials but supports them directly.

Meaningful types for multi-headed interfaces. The revised design supports special multi-headed interface types.

Examination of the undecidability proof in § 3 reveals that all types involved are (encodings of) interface types, thus subtyping remains undecidable even if regular interface types replace existentials. The real culprit for undecidability is the ability to provide implementation definitions for existentials or interface types. Moreover, such implementation definitions also prevent the assignment of minimal types to expressions, see Appendix C for an example.[7]

Hence, the revised design disallows implementation definitions for interface types. This restriction is rather severe because it prevents useful implementation definitions such as the one given in § 2.2, which implements `Iterable<T>` for all types implementing `List<T>`. *Abstract implementation definitions* are a possible cure. They look similar to regular implementation definitions but do not contribute to constraint entailment. Instead, they serve as blueprints for regular implementation definitions. Here is a revision of the example from § 2.2:

```
abstract implementation<X> Iterable<X> [List<X>] { /* body as before */ }
```

Regular implementation definitions may now inherit code from the abstract implementation. For example:

```
implementation<X> Iterable<X> [LinkedList<X>] extends [List<X>]
implementation<X> Iterable<X> [ArrayList<X>] extends [List<X>]
```

A disadvantage of abstract implementation definitions is that they do not induce a subtyping relation between the implementing type (`List<X>`) and the interface being implemented (`Iterable<X>`). While there is no problem for the concrete example (`List<X>` is a subinterface of `Iterable<X>` anyway), there are situations in which such a subtyping relation is desirable.

[7] Undecidability of subtyping in § 4 relies crucially on existentials with upper and lower bounds. If we removed lower bounds, then subtyping would become decidable. We do not consider this a viable option for JavaGI because it would require to add extra support for wildcards, leading to an overly complicated language design with existentials *and* wildcards.

6 Related Work

Kennedy and Pierce [10] investigate undecidability of subtyping under multiple instantiation inheritance and declaration-site variance. They prove that the general case is undecidable and present three decidable fragments. Our proof in § 3 is similar to theirs, although undecidability has different causes: Kennedy and Pierce's system is undecidable because of contravariant generic types, expansive class tables, and multiple instantiation inheritance, whereas undecidability of our system is due to the interaction of constraint entailment and subtyping caused by implementation definitions for existentials.

Pierce [14] proves undecidability of subtyping in $F_{\leq}$ by a chain of reductions from the halting problem for two-counter Turing machines. An intermediate link in this chain is the subtyping relation of $F^{D}_{\leq}$, which is also undecidable. Our proof in § 4 works by reduction from $F^{D}_{\leq}$ and is inspired by a reduction given by Ghelli and Pierce [6], who study bounded existential types in the context of $F_{\leq}$ and show undecidability of subtyping. Crucial to the undecidability proof of $F^{D}_{\leq}$ is rule D-ALL-NEG: it extends the typing context and essentially swaps the sides of a subtyping judgment. In $\mathcal{EX}_{uplo}$, rule S$_2$-OPEN and rule S$_2$-ABSTRACT together with lower bounds on type variables play a similar role.

Torgersen et al. [18] present WildFJ as a model for Java wildcards using existential types. The authors do not prove WildFJ sound. Cameron et al. [4] define a similar calculus ∃J and prove soundness. However, ∃J is not a full model for Java wildcards because it does not support lower bounds for type variables. The same authors present with TameFJ [3] a sound calculus supporting all essential features of Java wildcards. WildFJ's and TameFJ's subtyping rules are similar to the ones of $\mathcal{EX}_{uplo}$ defined in § 4, so the conjecture is that subtyping in WildFJ and TameFJ is also undecidable. The rule XS-ENV of TameFJ is roughly equivalent to the rules S$_2$-OPEN and S$_2$-ABSTRACT of $\mathcal{EX}_{uplo}$.

Decidability of subtyping for Java wildcards is still an open question [11]. One step in the right direction might be the work of Plümicke, who solves the problem of finding a substitution φ such that $\varphi T \leq \varphi U$ for Java types T, U with wildcards [15, 16]. Note that undecidability of $\mathcal{EX}_{uplo}$ does not imply undecidability for Java subtyping with wildcards. The proof of this claim would require a translation from subtyping derivations in $\mathcal{EX}_{uplo}$ to subtyping derivations in Java with wildcards, something we did not address in this article.

The programming language Scala [13] supports existential types in its latest release. The subtyping rules for existentials (§ 3.2.10 and § 3.5.2 of the specification [13]) are very similar to the ones for $\mathcal{EX}_{uplo}$. This raises the question whether Scala's subtyping relation with existentials is decidable.

7 Conclusion

The paper investigates decidability of subtyping with existential types in the context of JavaGI, Java wildcards, and Scala. In all cases, subtyping is undecidable. For JavaGI, there are some design options that avoid fully general existentials without giving up much expressivity.

Acknowledgments We thank the anonymous FTfJP reviewers for feedback on an earlier version of this article. We particularly thank the second reviewer for her/his numerous and extensive comments.

References

1. K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.
2. K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.
3. N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *22th European Conference on Object-Oriented Programming*, 2008. To appear.
4. N. Cameron, E. Ernst, and S. Drossopoulou. Towards an existential types model for Java wildcards. In *Workshop on Formal Techniques for Java-like Programs, informal proceedings*, 2007. `http://www.doc.ic.ac.uk/~ncameron/papers/cameron_ftfjp07_full.pdf`.
5. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–522, Dec. 1985.
6. G. Ghelli and B. Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1-2):75–96, 1998.
7. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, third edition, 2006.
8. M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK, 1994.
9. S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings 2nd European Symposium on Programming*, number 300 in Lecture Notes in Computer Science, pages 131–144. Springer-Verlag, 1988.
10. A. J. Kennedy and B. C. Pierce. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages, informal proceedings*, Jan. 2007. `http://foolwood07.cs.uchicago.edu/program/kennedy-abstract.html`.
11. K. Mazurak and S. Zdancewic. Type inference for Java 5: Wildcards, F-bounds, and undecidability. `http://www.cis.upenn.edu/~stevez/note.html`, 2006.
12. J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
13. M. Odersky. The Scala language specification version 2.7, Apr. 2008. Draft, `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.
14. B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.
15. M. Plümicke. Java type unification with wildcards. In *Proceedings of 17th International Conference on Applications of Declarative Programming and Knowledge Management and 21st Workshop on (Constraint) Logic Programming*, pages 234–245, Oct. 2007.
16. M. Plümicke. Typeless programming in Java 5.0 with wildcards. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. ACM, Sept. 2007.
17. E. L. Post. A variant of a recursivley unsolvable problem. *Bulletin of the American Mathematical Society*, 53:264–268, 1946.

18. M. Torgersen, E. Ernst, and C. P. Hansen. Wild FJ. In *International Workshop on Foundations of Object-Oriented Languages, informal proceedings*, 2005. `http://homepages.inf.ed.ac.uk/wadler/fool/program/14.html`.
19. M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, Dec. 2004.
20. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, Jan. 1989. ACM Press.
21. S. Wehr, R. Lämmel, and P. Thiemann. JavaGI: Generalized interfaces for Java. In E. Ernst, editor, *21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372, Berlin, Germany, July 2007. Springer-Verlag.
22. S. Wehr and P. Thiemann. Subtyping existential types. Technical Report 240, Universität Freiburg, June 2008. `ftp://ftp.informatik.uni-freiburg.de/documents/reports/report240/report00240.ps.gz`.

A Proof of Theorem 1

We first establish some auxiliary lemmas. The following lemma proves basic properties of our encoding scheme for words over Σ:

Lemma 1. *Suppose $u, v \in \Sigma^*$ and T is a type.*

(i) $[\![u]\!] = [\![v]\!]$ *iff* $u = v$
(ii) $u \mathbin{\#} (v \mathbin{\#} T) = uv \mathbin{\#} T$
(iii) $u \mathbin{\#} [\![v]\!] = [\![uv]\!]$

Proof. Straightforward. □

The next lemma ensures that the types occurring in a derivation of $\Theta_{\mathcal{P}}; \emptyset \vdash \exists\texttt{S<}[\![u_i]\!], [\![v_i]\!]\texttt{>} \leq \exists\texttt{G}$ are of a certain form. ($\Theta_{\mathcal{P}}$ is the program consisting of the interfaces and implementations defined in § 1.) We use the notation $[n]$ to denote the set $\{1, \dots, n\}$. Furthermore, $\mathfrak{I}$ and $\mathfrak{J}$ range over (possible empty) sequences of indices drawn from $[n]$, and $\mathfrak{I}\mathfrak{J}$ is the concatenation of $\mathfrak{I}$ and $\mathfrak{J}$. For $\mathfrak{I} = i_1 \dots i_r$, we write $u_{\mathfrak{I}}$ to denote the word $u_{i_1} \dots u_{i_r}$.

Lemma 2. *Suppose $\Theta_{\mathcal{P}}; \Delta \vdash T \leq W$. Let $\overline{U_i}^{i\in[k]}$ and $\overline{V_i}^{i\in[k]}$ be types such that $\mathsf{ftv}(\overline{U}, \overline{V}) = \emptyset$ and neither* $\texttt{S}$ *nor* $\texttt{G}$ *occur in $\overline{U}$ or $\overline{V}$. Assume*

$$T = \exists X\ \texttt{where}\ \overline{X\ \texttt{implements}\ \texttt{S<}U_i, V_i\texttt{>}}^{i\in[k]}\ .\ X \qquad \textit{or}$$

$$T = \exists X\ \texttt{where}\ X\ \texttt{implements}\ \texttt{G}, \overline{X\ \texttt{implements}\ \texttt{S<}U_i, V_i\texttt{>}}^{i\in[k]}\ .\ X \qquad \textit{or}$$

$$T = Z \text{ for some } Z \text{ with } Z \notin \mathsf{ftv}(W)$$

Moreover, assume that for all $Y\ \texttt{implements}\ I\texttt{<}\overline{W'}\texttt{>} \in \Delta$ either $I\texttt{<}\overline{W'}\texttt{>} = \texttt{G}$ or $I\texttt{<}\overline{W'}\texttt{>} = \texttt{S<}U_i, V_i\texttt{>}$ for some $i \in [k]$.

(i) Then $W = \exists X\ \texttt{where}\ \overline{P}\ .\ X$ and for all $P \in \overline{P}$ one of the following holds:
  (a) $P = X\ \texttt{implements}\ \texttt{S<}U_i, V_i\texttt{>}$ for some $i \in [k]$.
  (b) $P = X\ \texttt{implements}\ \texttt{S<}u_{\mathfrak{I}} \mathbin{\#} U_i, v_{\mathfrak{I}} \mathbin{\#} V_i\texttt{>}$ for some $i \in [k]$ and some non-empty sequence $\mathfrak{I}$.
  (c) $P = X\ \texttt{implements}\ \texttt{G}$.

(ii) If we additionally assume that

$$W = \exists X\ \texttt{where}\ X\ \texttt{implements}\ \texttt{G}, \overline{X\ \texttt{implements}\ \texttt{S<}U_i^*, V_i^*\texttt{>}}^{i\in[m]}\ .\ X$$

for some types $\overline{U^*}$ and $\overline{V^*}$, then one of the following holds:
  (a) $T = \exists X\ \texttt{where}\ X\ \texttt{implements}\ \texttt{G}, \overline{X\ \texttt{implements}\ \texttt{S<}U_i, V_i\texttt{>}}^{i\in[k]}\ .\ X$
  (b) there exists some $i \in [k]$ with $U_i = V_i$ or $u_{\mathfrak{I}} \mathbin{\#} U_i = v_{\mathfrak{I}} \mathbin{\#} V_i$ for some non-empty sequence $\mathfrak{I}$
  (c) $Y\ \texttt{implements}\ \texttt{G} \in \Delta$ for some Y

Proof. Both claims are proved by induction on the derivation of $\Theta_{\mathcal{P}}; \Delta \vdash T \leq W$.

(i) *Case distinction* on the last rule used.

- *Case* rule S1-REFL: Then $T = W$. The case $T = Z$ for some Z is not possible because we would then also have that $Z \in \mathsf{ftv}(W)$. For the other cases, the claim follows trivially.
- *Case* rule S1-TRANS: Then

$$\frac{\Theta_{\mathcal{P}}; \Delta \vdash T \leq V \qquad \Theta_{\mathcal{P}}; \Delta \vdash V \leq W}{\Theta_{\mathcal{P}}; \Delta \vdash T \leq W}$$

Applying the I.H. to $\Theta_{\mathcal{P}}; \Delta \vdash T \leq V$ gives us $V = \exists Y\, \texttt{where}\, \overline{Q_j}^{j \in [m]} \,.\, Y$ such that for all Q_j one of the following holds:

(a) $Q_j = Y\, \texttt{implements}\, \texttt{S<}U_{\varphi(j)}, V_{\varphi(j)}\texttt{>}$ for some $\varphi(j) \in [k]$.

(b) $Q_j = Y\, \texttt{implements}\, \texttt{S<}u_{\mathfrak{I}_j} \# U_{\varphi(i)}, v\#_{\mathfrak{I}_j} V_{\varphi(i)}\texttt{>}$ for some $\varphi(j) \in [k]$ and some non-empty sequence $\mathfrak{I}_j$.

(c) $Q_j = Y\, \texttt{implements}\, \texttt{G}$.

Define for all $j \in \{j' \in [m] \mid \text{(a) holds for } j'\}$

$$U'_j = U_{\varphi(j)}$$
$$V'_j = V_{\varphi(j)}$$

and for all $j \in \{j' \in [m] \mid \text{(b) holds for } j'\}$

$$U'_j = u_{\mathfrak{I}_j} \# U_{\varphi(j)}$$
$$V'_j = v_{\mathfrak{I}_j} \# V_{\varphi(j)}$$

Then define

$$\overline{U''_i}^{i \in \mathcal{M}} = \overline{U}, \overline{U'_j}^{j \text{ such that } U'_j \text{ defined}}$$
$$\overline{V''_i}^{i \in \mathcal{M}} = \overline{V}, \overline{V'_j}^{j \text{ such that } V'_j \text{ defined}}$$

Clearly $\mathsf{ftv}(\overline{U''}, \overline{V''}) = \emptyset$ and neither $\texttt{S}$ nor $\texttt{G}$ occur in $\overline{U''}, \overline{V''}$. Moreover, V and Δ have the right form to apply to I.H. with $\overline{U''}, \overline{V''}$. We then get $W = \exists X\, \texttt{where}\, \overline{P} \,.\, X$ and for all $P \in \overline{P}$ one of the following holds:

(a) $P = X\, \texttt{implements}\, \texttt{S<}U''_i, V''_i\texttt{>}$ for some $i \in \mathcal{M}$.

(b) $P = X\, \texttt{implements}\, \texttt{S<}u_{\mathfrak{I}} \# U''_i, v_{\mathfrak{I}} \# V''_i\texttt{>}$ for some $i \in \mathcal{M}$ and some non-empty sequence $\mathfrak{I}$.

(c) $P = X\, \texttt{implements}\, \texttt{G}$.

In all three cases, we can easily verify that P is of the right form. In case (b), we need to use Lemma 1(ii).

- *Case* rule S1-OPEN: Then

$$\frac{\Theta_{\mathcal{P}}; \Delta, \overline{Q} \vdash Y \leq W \qquad Y \notin \mathsf{ftv}(W)}{\Theta_{\mathcal{P}}; \Delta \vdash \underbrace{\exists Y\, \texttt{where}\, \overline{Q} \,.\, Y}_{=T} \leq W}$$

and the claim follows by applying the I.H. to $\Theta_{\mathcal{P}}; \Delta, \overline{Q} \vdash Y \leq T$.

- *Case* rule S$_1$-ABSTRACT: Then

  $$\frac{(\forall l)\ \Theta_{\mathcal{P}}; \Delta \Vdash [T/X]P_l}{\Theta_{\mathcal{P}}; \Delta \vdash T \leq \underbrace{\exists X\,\texttt{where}\,\overline{P}\,.\,X}_{=W}}$$

  Assume $P \in \overline{P}$. Then P has the form $X\,\texttt{implements}\,\ldots$
  Case distinction on the rule used to derive $\Theta_{\mathcal{P}}, \Delta \Vdash P$.
  - *Case* rule E$_1$-LOCAL: Then $[T/X]P \in \Delta$. Hence, either $[T/X]P = T\,\texttt{implements}\,\texttt{G}$ or $[T/X]P = T\,\texttt{implements}\,\texttt{S<}U_i, V_i\texttt{>}$ for $i \in [k]$. As $\mathsf{ftv}(\overline{U}, \overline{V}) = \emptyset$ by assumption, we have either $P = X\,\texttt{implements}\,\texttt{G}$ or $P = X\,\texttt{implements}\,\texttt{S<}U_i, V_i\texttt{>}$ as required.
  - *Case* rule E$_1$-IMPL: There are two possibilities:
    * $\texttt{implementation<}X, Y\texttt{>}\ \texttt{S<}u_j \# X, v_j \# Y\texttt{>}\ \texttt{[}\exists\texttt{S<}X, Y\texttt{>]} \in \Theta_{\mathcal{P}}$ and $[T/X]P = \exists\texttt{S<}U', V'\texttt{>}\,\texttt{implements}\,\texttt{S<}u_j\#U', v\#_j V'\texttt{>}$. Then $T = \exists\texttt{S<}U', V'\texttt{>}$, so $U' = U_i$ and $V' = V_i$ for some $i \in [k]$. With $\mathsf{ftv}(\overline{U}, \overline{V}) = \emptyset$ also $P = X\,\texttt{implements}\,\texttt{S<}u_j \# U_i, v_j \# V_i\texttt{>}$ as required.
    * $\texttt{implementation<}X\texttt{>}\ \texttt{G}\ \texttt{[}\exists\texttt{S<}X, X\texttt{>]} \in \Theta_{\mathcal{P}}$ and $[T/X]P = \exists\texttt{S<}U', V'\texttt{>}\,\texttt{implements}\,\texttt{G}$. But then also $P = X\,\texttt{implements}\,\texttt{G}$.

  End case distinction on the rule used to derive $\Theta_{\mathcal{P}}, \Delta \Vdash P$.

End case distinction on the last rule used.

(ii) *Case distinction* on the last rule used.

- *Case* rule S$_1$-REFL: Trivial.
- *Case* rule S$_1$-TRANS: Then

  $$\frac{\Theta_{\mathcal{P}}; \Delta \vdash T \leq V \qquad \Theta_{\mathcal{P}}; \Delta \vdash V \leq W}{\Theta_{\mathcal{P}}; \Delta \vdash T \leq W}$$

  We now apply part (i) of this lemma to $\Theta_{\mathcal{P}}; \Delta \vdash T \leq V$ and get that $V = \exists Y\,\texttt{where}\,\overline{Q_j}^{\,j \in [m]}\,.\,Y$ such that for all Q_j either (a), (b), or (c) as in case S$_1$-TRANS of the proof for part (i) holds. Define $\overline{U''_i}^{\,i \in \mathcal{M}}$ and $\overline{V''_i}^{\,i \in \mathcal{M}}$ as in case S$_1$-TRANS of the proof for part (i). For $\overline{U''}, \overline{V''}$, we now can apply the I.H. of this part of the proof to $\Theta_{\mathcal{P}}; \Delta \vdash V \leq W$ and get that one of the following holds:

  (a) $V = \exists Y\,\texttt{where}\,Y\,\texttt{implements}\,\texttt{G}, \overline{Y\,\texttt{implements}\,\texttt{S<}U''_i, V''_i\texttt{>}}^{\,i \in \mathcal{M}}\,.\,Y$. Then the claim follows by applying the I.H. to $\Theta_{\mathcal{P}}; \Delta \vdash T \leq V$.

  (b) there exists some $i \in \mathcal{M}$ with $U''_i = V''_i$ or $u_{\mathfrak{J}} \# U''_i = v_{\mathfrak{J}} \# V''_i$ for some non-empty sequence $\mathfrak{J}$.
  - If $U''_i = U_j$ for some $j \in [k]$ then $V''_i = V_j$ and the claim is immediate.
  - Otherwise, $U''_i = U'_j$ and $V''_i = V'_j$ for some $j \in [m]$.
    * If now $U'_j = U_{\varphi(j)}$ then $V'_j = V_{\varphi(j)}$ and the claim is immediate.

        - ∗ Otherwise $U'_j = u_{\mathfrak{I}_j} \mathbin{\#} U_{\varphi(j)}$ and $V'_j = v_{\mathfrak{I}_j} \mathbin{\#} V_{\varphi(j)}$ for some non-empty sequence $\mathfrak{I}_j$. If $U''_i = V''_i$ then the claim is immediate; otherwise, we need to use Lemma 1(ii).
    - (c) Y `implements` `G` $\in \Delta$ for some Y. The claim then holds trivially.
- *Case* rule S₁-OPEN: Then

$$\frac{\Theta_{\mathcal{P}}; \Delta, \overline{Q} \vdash Y \leq W \qquad Y \notin \mathsf{ftv}(W)}{\Theta_{\mathcal{P}}; \Delta \vdash \underbrace{\exists Y \,\texttt{where}\, \overline{Q}\,.\, Y}_{=T} \leq W}$$

  and the claim follows by applying the I.H. to $\Theta_{\mathcal{P}}; \Delta, \overline{Q} \vdash Y \leq T$.
- *Case* rule S₁-ABSTRACT: Then

$$\frac{(\forall l)\ \Theta_{\mathcal{P}}; \Delta \Vdash [T/X]P_l}{\Theta_{\mathcal{P}}; \Delta \vdash T \leq \underbrace{\exists X \,\texttt{where}\, \overline{P}\,.\, X}_{=W}}$$

  We have X `implements` `G` $\in \overline{P}$ so

$$\Theta_{\mathcal{P}}; \Delta \Vdash T \,\texttt{implements}\, \texttt{G}$$

  Case distinction on the rule used to derive $\Theta_{\mathcal{P}}, \Delta \Vdash T$ `implements` `G`.
  - *Case* rule E₁-LOCAL: Then T `implements` `G` $\in \Delta$. Hence, $T = Z$ and the claim holds.
  - *Case* rule E₁-IMPL: Then implementation definition (1) must have been used in the premise of the rule. Hence, $T = \exists\texttt{S<}U', U'\texttt{>}$, so $U_1 = V_1$ as required.

  End case distinction on the rule used to derive $\Theta_{\mathcal{P}}, \Delta \Vdash T$ `implements` `G`.

End case distinction on the last rule used. □

Finally, we prove the following lemma which directly implies Theorem 1.

Lemma 3. *The PCP instance $\mathcal{P} = \{(u_1, v_1), \ldots, (u_n, v_n)\}$ has a solution if and only if there exists some $i \in \{1, \ldots, n\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash \exists\texttt{S<}\llbracket u_i \rrbracket, \llbracket v_i \rrbracket\texttt{>} \leq \exists\texttt{G}$ is derivable.*

Proof.

"⇒": We first show for any non-empty sequence of indices $i_1 \ldots i_k$ that

$$\Theta_{\mathcal{P}}; \emptyset \vdash \exists S\texttt{<}\llbracket u_{i_k} \rrbracket, \llbracket v_{i_k} \rrbracket\texttt{>} \leq \exists S\texttt{<}\llbracket u_{i_1} \ldots u_{i_k} \rrbracket, \llbracket v_{i_1} \ldots v_{i_k} \rrbracket\texttt{>} \tag{3}$$

The proof is by induction on k. The base case follows from reflexivity of subtyping. For the inductive step, the induction hypothesis yields

$$\begin{gathered} \Theta_{\mathcal{P}}; \emptyset \vdash \exists S\texttt{<}\llbracket u_{i_{k+1}} \rrbracket, \llbracket v_{i_{k+1}} \rrbracket\texttt{>} \leq T \\ \text{where } T = \exists S\texttt{<}\llbracket u_{i_2} \ldots u_{i_{k+1}} \rrbracket, \llbracket v_{i_2} \ldots v_{i_{k+1}} \rrbracket\texttt{>} \quad . \end{gathered} \tag{4}$$

E2-EXTENDS’ $$\frac{\mathit{\Delta} \vdash' T \leq U}{\mathit{\Delta} \Vdash' T\, \texttt{extends}\, U}$$

E2-SUPER’ $$\frac{\mathit{\Delta} \vdash' U \leq T}{\mathit{\Delta} \Vdash' T\, \texttt{super}\, U}$$

S2-REFL’ $$\frac{T = X \text{ or } T = N}{\mathit{\Delta} \vdash' T \leq T}$$

S2-OBJECT’ $$\mathit{\Delta} \vdash' T \leq \texttt{Object}$$

S2-EXTENDS’ $$\frac{X\, \texttt{extends}\, T' \in \mathit{\Delta} \qquad \mathit{\Delta} \vdash' T' \leq T}{\mathit{\Delta} \vdash' X \leq T}$$

S2-SUPER’ $$\frac{X\, \texttt{super}\, T' \in \mathit{\Delta} \qquad \mathit{\Delta} \vdash' T \leq T'}{\mathit{\Delta} \vdash' T \leq X}$$

S2-OPEN’ $$\frac{\mathit{\Delta}, \overline{P} \vdash' N \leq T \qquad \overline{X} \cap \mathsf{ftv}(\mathit{\Delta}, T) = \emptyset}{\mathit{\Delta} \vdash' \exists \overline{X}\, \texttt{where}\, \overline{P} \,.\, N \leq T}$$

S2-ABSTRACT’ $$\frac{N = [\overline{Y/X}]M \qquad (\forall i)\ \mathit{\Delta} \Vdash' [\overline{Y/X}]P_i}{\mathit{\Delta} \vdash' N \leq \exists \overline{X}\, \texttt{where}\, \overline{P} \,.\, M}$$

Fig. 5. Subtyping for $\mathcal{EX}_{uplo}$without transitivity rule

Choosing a suitable implementation definition from (1), we get

$$\mathit{\Theta}_{\mathcal{P}}; \emptyset \Vdash T\, \texttt{implements}\, \texttt{S<}u_{i_1} \# \llbracket u_{i_2} \dots u_{i_{k+1}} \rrbracket, v_{i_1} \# \llbracket v_{i_2} \dots v_{i_{k+1}} \rrbracket\texttt{>}$$

Hence, by Lemma 1(iii) and rule S1-ABSTRACT we also have

$$\mathit{\Theta}_{\mathcal{P}}; \emptyset \vdash T \leq \exists S\texttt{<}\llbracket u_{i_1} \dots u_{i_{k+1}} \rrbracket, \llbracket v_{i_1} \dots v_{i_{k+1}} \rrbracket\texttt{>}$$

Claim (3) now follows with (4) and transitivity of subtyping.

Now suppose that $\mathfrak{I} = i_1 \dots i_r$ is a solution to $\mathcal{P}$. Then we have from (3)

$$\mathit{\Theta}_{\mathcal{P}}; \emptyset \vdash \exists S\texttt{<}\llbracket u_{i_r} \rrbracket, \llbracket v_{i_r} \rrbracket\texttt{>} \leq \exists S\texttt{<}\llbracket u_{\mathfrak{I}} \rrbracket, \llbracket v_{\mathfrak{I}} \rrbracket\texttt{>}$$

Because $u_{\mathfrak{I}} = v_{\mathfrak{I}}$ we get $\llbracket u_{\mathfrak{I}} \rrbracket = \llbracket v_{\mathfrak{I}} \rrbracket$ by Lemma 1(i), so implementation definition (1) yields

$$\mathit{\Theta}_{\mathcal{P}}; \emptyset \Vdash \exists S\texttt{<}\llbracket u_{\mathfrak{I}} \rrbracket, \llbracket v_{\mathfrak{I}} \rrbracket\texttt{>}\, \texttt{implements}\, \texttt{G}$$

Applying rules S1-ABSTRACT and S1-TRANS gives us $\mathit{\Theta}_{\mathcal{P}}; \emptyset \vdash \exists S\texttt{<}\llbracket u_{i_r} \rrbracket, \llbracket v_{i_r} \rrbracket\texttt{>} \leq \exists\texttt{G}$, as required.

“$\Leftarrow$”: Given that $\mathit{\Theta}_{\mathcal{P}}; \emptyset \vdash \exists\texttt{S<}\llbracket u_i \rrbracket, \llbracket v_i \rrbracket\texttt{>} \leq \exists\texttt{G}$ is derivable for some $i \in \{1, \dots, n\}$, we get from Lemma 2(ii) that either $\llbracket u_i \rrbracket = \llbracket v_i \rrbracket$ or that there exists a non-empty sequence $\mathfrak{I}$ such that $u_{\mathfrak{I}} \# \llbracket u_i \rrbracket = v_{\mathfrak{I}} \# \llbracket v_i \rrbracket$. For the first case, we have $u_i = v_i$ by Lemma 1(i); for the second case, we get $\llbracket u_{\mathfrak{I}} u_i \rrbracket = \llbracket v_{\mathfrak{I}} v_i \rrbracket$ by Lemma 1(iii), and $u_{\mathfrak{I}} u_i = v_{\mathfrak{I}} v_i$ by Lemma 1(i). Hence, $\mathcal{P}$ has a solution. □

B Proof of Theorem 2

Fig. 5 shows the definition of the alternative subtyping relation $\mathit{\Delta} \vdash' T \leq T$ for $\mathcal{EX}_{uplo}$ that does not have a built-in transitivity rule. To establish equivalence with the original subtyping relation (Fig. 2), we first prove that the alternative subtyping relation is reflexive and transitive.

Lemma 4 (Reflexivity). *For all types T, $\Delta \vdash' T \le T$.*

Proof. The only interesting case is $T = \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N$. Then we have

$$\textsc{s}_2\text{-OPEN'}\ \frac{\textsc{s}_2\text{-ABSTRACT'}\ \dfrac{N = N \qquad (\forall i)\ \Delta, \overline{P} \Vdash' P_i}{\Delta, \overline{P} \vdash' N \le \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N} \qquad \overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset}{\Delta \vdash' \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N \le \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N}$$

Note that it is easy to verify that $\Delta \Vdash' P$ for any $P \in \Delta$. □

Lemma 5 (Transitivity). *If $\Delta \vdash' T \le U$ and $\Delta \vdash' U \le V$, then $\Delta \vdash' T \le V$.*

The prove of the transitivity lemma makes essential use of the fact that type variables do not have both lower and upper bounds and that only type variables may occur as type arguments of generic classes.

Proof. We define the *size* of a type or constraint as follows:

$$\begin{aligned}
\mathsf{size}(X) &= 1 \\
\mathsf{size}(C\texttt{<}\overline{T}\texttt{>}) &= 1 + \mathsf{size}(\overline{T}) \\
\mathsf{size}(\texttt{Object}) &= 1 \\
\mathsf{size}(\exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N) &= 1 + \mathsf{size}(\overline{P}) + \mathsf{size}(N) \\
\mathsf{size}(X\,\texttt{extends}\,T) &= \mathsf{size}(T) \\
\mathsf{size}(X\,\texttt{super}\,T) &= \mathsf{size}(T)
\end{aligned}$$

We use the notation $\mathsf{size}(\overline{\xi})$ as an abbreviation for $\sum_i \mathsf{size}(\xi_i)$.

Moreover, we define the *domain* of a type environment Δ as $\mathsf{dom}(\Delta) = \{X \mid X\,\texttt{extends}\,T \in \Delta \text{ or } X\,\texttt{super}\,T \in \Delta\}$, and the *range* of a type environment Δ as $\mathsf{rng}(\Delta) = \{T \mid X\,\texttt{extends}\,T \in \Delta \text{ or } X\,\texttt{super}\,T \in \Delta\}$.

We now strengthen the claim as follows:

> *Let $n \in \mathbb{N}$.*
>
> *(i) Assume $\mathsf{size}(U) = n$. If $\Delta \vdash' T \le U$ and $\Delta \vdash' U \le V$, then $\Delta \vdash' T \le V$.*
>
> *(ii) Assume $\mathsf{size}(\overline{P}) = n$. If $\Delta', \overline{P} \vdash' W_1 \le W_2$ and $[\overline{Y/X}]\Delta' \Vdash' [\overline{Y/X}]P$ for all $P \in \overline{P}$ and $\overline{X} \cap \mathsf{dom}(\Delta') = \emptyset$, then $[\overline{Y/X}]\Delta' \vdash' [\overline{Y/X}]W_1 \le [\overline{Y/X}]W_2$.*

We then prove that claims (i) and (ii) hold for all $n \in \mathbb{N}$ by complete induction. Suppose $n \in \mathbb{N}$ and assume that

$$\text{(i) and (ii) hold for all } n' \in \mathbb{N} \text{ with } n' < n. \tag{5}$$

We now have to prove that (i) and (ii) hold for n.

(i) We prove claim (i) by induction on the combined size of the derivations of $\Delta \vdash' T \leq U$ and $\Delta \vdash' U \leq V$. We perform a case analysis on the last rules used in these derivations. The following tables lists all possible combinations; the rows contain the last rule used in $\Delta \vdash' T \leq U$, the columns the last rule used in $\Delta \vdash' U \leq V$.

	S$_2$-REFL’	S$_2$-OBJECT’	S$_2$-EXTENDS’	S$_2$-SUPER’	S$_2$-OPEN’	S$_2$-ABSTRACT’
S$_2$-REFL’	✓	✓	✓	✓	✓	✓
S$_2$-OBJECT’	✓	✓	↯	✓	↯	(a)
S$_2$-EXTENDS’	✓	✓	✓	✓	✓	✓
S$_2$-SUPER’	✓	✓	(b)	✓	↯	↯
S$_2$-OPEN’	✓	✓	✓	✓	✓	✓
S$_2$-ABSTRACT’	✓	✓	↯	✓	(c)	↯

Cases marked with ✓ are trivial or follow directly from the inner induction hypothesis; cases marked with ↯ can never occur because they put conflicting constraints on the form of U. We now deal with the remaining cases.

(a) Then $U = \mathtt{Object}$, $V = \exists \overline{X}\ \mathtt{where}\ \overline{P} \,.\, N$ and the premise of S$_2$-ABSTRACT’ requires $\mathtt{Object} = [\overline{Y/X}]N$, so $N = \mathtt{Object}$. But this contradicts the restriction in restriction (1) on page 6.

(b) Then $U = X$ and Δ contains an lower and upper bound for X. This is a contradiction to restriction (3) on page 6.

(c) Then $T = M$ and $U = \exists \overline{X}\ \mathtt{where}\ \overline{P} \,.\, N$ and

$$\frac{\begin{array}{c} M = [\overline{Y/X}]N \\ (\forall i)\ \Delta \Vdash' [\overline{Y/X}]P_i \end{array}}{\Delta \vdash' M \leq \exists \overline{X}\ \mathtt{where}\ \overline{P} \,.\, N} \qquad \frac{\begin{array}{c} \Delta, \overline{P} \vdash' N \leq V \\ \mathsf{ftv}(\Delta, V) \cap \overline{X} = \emptyset \end{array}}{\Delta \vdash' \exists \overline{X}\ \mathtt{where}\ \overline{P} \,.\, N \leq V}$$

We have

$$\mathsf{size}(\overline{P}) < \mathsf{size}(U) = n$$

With (5) we then get

$$[\overline{Y/X}]\Delta \vdash' [\overline{Y/X}]N \leq [\overline{Y/X}]V$$

Because $T = [\overline{Y/X}]N$ and $\overline{X} \cap \mathsf{ftv}(\Delta, V) = \emptyset$, we have

$$\Delta \vdash' T \leq V$$

as required.

(ii) We proceed by induction on the derivation $\mathcal{D}$ of $\Delta', \overline{P} \vdash' W_1 \leq W_2$. We have already proved (i) for n, so with (5)

$$\text{(i) holds for all } n' \in \mathbb{N} \text{ with } n' \leq n \tag{6}$$

Case distinction on the last rule used in $\mathcal{D}$.

– *Case* rule S$_2$-REFL’: Follows with Lemma 4.

- *Case* rule S2-OBJECT': Trivial.
- *Case* rule S2-EXTENDS': We then have $W_1 = X$ and

  $$\frac{X\,\texttt{extends}\,W_2' \in \Delta', \overline{P} \qquad \Delta', \overline{P} \vdash' W_2' \leq W_2}{\Delta', \overline{P} \vdash' X \leq W_2}$$

  Applying the inner I.H. yields

  $$[\overline{Y/X}]\Delta' \vdash' [\overline{Y/X}]W_2' \leq [\overline{Y/X}]W_2 \tag{7}$$

  - If $X\,\texttt{extends}\,W_2' \in \overline{P}$ then

    $$[\overline{Y/X}]\Delta' \vdash' [\overline{Y/X}]X \leq [\overline{Y/X}]W_2' \tag{8}$$

    by the assumption. We also have

    $$\mathsf{size}([\overline{Y/X}]W_2') = \mathsf{size}(W_2') \leq \mathsf{size}(\overline{P}) = n$$

    Using (6) on (8) and (7) yields

    $$[\overline{Y/X}]\Delta' \vdash' [\overline{Y/X}]X \leq [\overline{Y/X}]W_2$$

    as required.
  - If $X\,\texttt{extends}\,W_2' \in \Delta'$ then $[\overline{Y/X}]X = X$ because $\overline{X} \cap \mathsf{dom}(\Delta) = \emptyset$. With (7) and rule S2-EXTENDS', we get the required result.
- *Case* rule S2-SUPER': Follows analogously.
- *Case* rule S2-OPEN': Then $W_1 = \exists\overline{Z}\,\texttt{where}\,\overline{Q}\,.\,N$ and

  $$\frac{\Delta', \overline{P}, \overline{Q} \vdash' N \leq W_2 \qquad \overline{Z} \cap \mathsf{ftv}(\Delta', \overline{P}, W_2) = \emptyset}{\Delta', \overline{P} \vdash' \exists\overline{Z}\,\texttt{where}\,\overline{Q}\,.\,N \leq W_2}$$

  Because the $\overline{Z}$ are sufficiently fresh, we may assume

  $$[\overline{Y/X}](\exists\overline{Z}\,\texttt{where}\,\overline{Q}\,.\,N) = \exists\overline{Z}\,\texttt{where}\,([\overline{Y/X}]\overline{Q})\,.\,([\overline{Y/X}]N)$$
  $$\overline{Z} \cap \mathsf{ftv}([\overline{Y/X}]\Delta, [\overline{Y/X}]W_2) = \emptyset$$

  Using the inner I.H. yields

  $$[\overline{Y/X}](\Delta', \overline{Q}) \vdash' [\overline{Y/X}]N \leq [\overline{Y/X}]W_2$$

  Thus with S2-OPEN'

  $$[\overline{Y/X}]\Delta' \vdash' [\overline{Y/X}](\exists\overline{Z}\,\texttt{where}\,\overline{Q}\,.\,N) \leq [\overline{Y/X}]W_2$$

- *Case* rule S2-ABSTRACT': Then $W_2 = \exists\overline{Z}\,\texttt{where}\,\overline{Q}\,.\,N$ and

  $$\frac{W_1 = [\overline{Y'/Z}]N \qquad (\forall i)\ \Delta', \overline{P} \Vdash' [\overline{Y'/Z}]Q_i}{\Delta', \overline{P} \vdash' W_1 \leq \exists\overline{Z}\,\texttt{where}\,\overline{Q}\,.\,N}$$

Using the inner I.H., we can easily verify that

$$(\forall i)\ [\overline{Y/X}]\Delta' \Vdash' [\overline{Y/X}][\overline{Y'/Z}]Q_i$$

Because the $\overline{Z}$ are sufficiently fresh, we may assume

$$\begin{gathered}[\overline{Y/X}](\exists\overline{Z}\,\texttt{where}\,\overline{Q}\,.\,N) = \exists\overline{Z}\,\texttt{where}\,([\overline{Y/X}]\overline{Q})\,.\,([\overline{Y/X}]N)\\ \overline{Z}\cap\overline{Y} = \emptyset\end{gathered}$$

Moreover, for $\varphi = [\overline{[\overline{Y/X}]Y'/Z}]$, we have

$$\begin{gathered}[\overline{Y/X}][\overline{Y'/Z}]N = \varphi[\overline{Y/X}]N\\ [\overline{Y/X}][\overline{Y'/Z}]\overline{Q} = \varphi[\overline{Y/X}]\overline{Q}\end{gathered}$$

Hence,

$$\begin{gathered}[\overline{Y/X}]W_1 = \varphi[\overline{Y/X}]N\\ (\forall i)\ [\overline{Y/X}]\Delta' \Vdash' \varphi[\overline{Y/X}]Q_i\end{gathered}$$

The claim now follows with rule S$_2$-ABSTRACT’.

End case distinction on the last rule used in $\mathcal{D}$. □

Now we can prove that $\Delta \vdash T \leq U$ and $\Delta \vdash' T \leq U$ coincide.

Lemma 6 (Soundness and completeness). *$\Delta \vdash T \leq U$ if and only if $\Delta \vdash' T \leq U$.*

Proof. Both directions of the lemma are proved by a straightforward induction on the derivation given. For the “⇒” direction, we note two things:

– When the derivation of $\Delta \vdash T \leq U$ ends with rule S$_2$-TRANS, we apply the I.H. to the two subderivations and combine the two resulting derivations using Lemma 5.

– When the derivation of $\Delta \vdash T \leq U$ ends with rule S$_2$-ABSTRACT, we have $N = [\overline{T/X}]M$ as a premise. But the corresponding rule S$_2$-ABSTRACT’ requires $N = [\overline{Y/X}]M$. We can easily show $\overline{T} = \overline{Y}$ for some $\overline{Y}$ because N has the form $C\texttt{<}\overline{Z}\texttt{>}$ (see the syntax in Fig. 2). □

Our next goal is to show that $[\![\Gamma]\!]^- \vdash' [\![\tau]\!]^- \leq [\![\tau']\!]^+$ implies $\Gamma \vdash \tau \leq \tau'$. Before proving this fact, we need to establish some more lemmas. In the following, we use the notation $\mathcal{D} :: \mathcal{J}$ to denote that $\mathcal{D}$ is a derivation for judgment $\mathcal{J}$ and define $\mathsf{height}(\mathcal{D})$ as the height of $\mathcal{D}$.

Lemma 7 (Strengthening). *Suppose $X \notin \mathsf{ftv}(\Delta, T, U, V)$. If $\mathcal{D} :: \Delta, X\,\texttt{super}\,T \vdash' U \leq V$ or $\mathcal{D} :: \Delta, X\,\texttt{extends}\,T \vdash' U \leq V$, then $\mathcal{D} :: \Delta \vdash' U \leq V$ with $\mathsf{height}(\mathcal{D}) = \mathsf{height}(\mathcal{D}')$.*

Proof. Straightforward induction on $\mathcal{D}$. □

Lemma 8.

(i) If $\mathcal{D} :: \Delta, X\, \texttt{super}\, T \vdash' U \leq X$ *with* $X \notin \mathsf{ftv}(\Delta, T, U)$*, then* $\mathcal{D}' :: \Delta \vdash' U \leq T$ *with* $\mathsf{height}(\mathcal{D}') \leq \mathsf{height}(\mathcal{D})$.

(ii) If $\mathcal{D} :: \Delta, X\, \texttt{extends}\, T \vdash' X \leq U$ *with* $X \notin \mathsf{ftv}(\Delta, T, U)$*, then* $\mathcal{D}' :: \Delta \vdash' T \leq U$ *with* $\mathsf{height}(\mathcal{D}') \leq \mathsf{height}(\mathcal{D})$.

Proof.

(i) Induction on $\mathcal{D}$.
Case distinction on the last rule of $\mathcal{D}$.

- *Case* rule S2-REFL': Impossible.
- *Case* rule S2-OBJECT': Impossible.
- *Case* rule S2-EXTENDS': Follows by I.H. and rule S2-EXTENDS'.
- *Case* rule S2-SUPER': Then $\Delta, X\, \texttt{super}\, T \vdash' U \leq T$ from the premise and the claim follows with Lemma 7.
- *Case* rule S2-OPEN': Then $U = \exists \overline{Y}\, \texttt{where}\, \overline{Q}\, .\, N$ and

$$\textsc{s}_2\text{-OPEN'}\ \dfrac{\textsc{s}_2\text{-SUPER'}\ \dfrac{\mathcal{D}_1 :: \Delta, X\, \texttt{super}\, T, \overline{Q} \vdash' N \leq T}{\Delta, X\, \texttt{super}\, T, \overline{Q} \vdash' N \leq X} \qquad \overline{Y} \cap \mathsf{ftv}(\Delta, X, T) = \emptyset}{\mathcal{D} :: \Delta, X\, \texttt{super}\, T \vdash' \exists \overline{Y}\, \texttt{where}\, \overline{Q}\, .\, N \leq X}$$

We have $X \notin \mathsf{ftv}(\overline{Q}, N)$ because $X \notin \mathsf{ftv}(U)$. With Lemma 7

$$\mathcal{D}_1' :: \Delta, \overline{Q} \vdash' N \leq T$$
$$\mathsf{height}(\mathcal{D}_1) = \mathsf{height}(\mathcal{D}_1')$$

The claim now follows with rule S2-OPEN'.
- *Case* rule S2-ABSTRACT': Impossible.

End case distinction on the last rule of $\mathcal{D}$.

(ii) *Case distinction* on the last rule of $\mathcal{D}$.

- *Case* rule S2-REFL': Impossible.
- *Case* rule S2-OBJECT': Trivial.
- *Case* rule S2-EXTENDS': Then $\Delta, X\, \texttt{extends}\, T \vdash' T \leq U$ from the premise and the claim follows with Lemma 7.
- *Case* rule S2-SUPER': Follows by I.H. and rule S2-SUPER'.
- *Case* rule S2-OPEN': Impossible.
- *Case* rule S2-ABSTRACT': Impossible.

End case distinction on the last rule of $\mathcal{D}$. □

Lemma 9. *Let* τ^- *and* σ^+ *be* $F_{\leq}^{D}$ *types. Then* $[\![\tau]\!]^- \neq [\![\sigma]\!]^+$.

Proof. Obvious. □

Lemma 10. *If* $[\![\Gamma]\!]^- \vdash' [\![\tau]\!]^- \leq [\![\tau']\!]^+$ *then* $\Gamma \vdash \tau \leq \tau'$.

Proof. Let $[\![\Gamma]\!]^- = \Delta$, $[\![\tau]\!]^- = T$, and $[\![\tau']\!]^+ = U$. Proceed by induction on the given derivation.
Case distinction on the last rule of this derivation.

- *Case* rule $\textsc{s}_2$-REFL': Then $T = U$ so $\llbracket\tau\rrbracket^- = \llbracket\tau'\rrbracket^+$ which is impossible by Lemma 9.
- *Case* rule $\textsc{s}_2$-OBJECT': Then $\tau' = \textsf{Top}$ and the claim follows by D-TOP.
- *Case* rule $\textsc{s}_2$-EXTENDS': Then $T = X^\alpha$ and $\tau = \alpha$ and

$$\frac{X\,\texttt{extends}\,T' \in \Delta \qquad \Delta \vdash' T' \leq U}{\Delta \vdash' X^\alpha \leq U}$$

Because $\Delta = \llbracket\Gamma\rrbracket^-$, we have $T' = \llbracket\sigma\rrbracket^-$ and $\Gamma(\alpha) = \sigma^-$. Applying the I.H. yields

$$\Gamma \vdash \sigma \leq \tau'$$

so the claim follows by rule D-VAR.

- *Case* rule $\textsc{s}_2$-SUPER': Impossible because n-positive types are not variables.
- *Case* rule $\textsc{s}_2$-OPEN': Hence $T = \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N$ and

$$\frac{\Delta, \overline{P} \vdash' N \leq T \qquad \overline{X} \cap \textsf{ftv}(\Delta, U) = \emptyset}{\Delta \vdash' \exists\overline{X}\,\texttt{where}\,\overline{P}\,.\,N \leq U}$$

From $T = \llbracket\tau\rrbracket^-$ we have

$$\begin{aligned}
\tau &= \forall\alpha_0 \ldots \alpha_n\,.\,\neg\,\sigma \\
T &= \neg\,\overbrace{\begin{array}{l}\exists X^{\alpha_0} \ldots X^{\alpha_n}\,Y\,\texttt{where}\,Y\,\texttt{extends}\,\llbracket\sigma\rrbracket^+ \\ \qquad\qquad\qquad .\,C^{n+2}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>}\end{array}}^{=T'} \\
&= \exists X\,\texttt{where}\,X\,\texttt{super}\,T'\,.\,D^{1}\texttt{<}X\texttt{>}
\end{aligned}$$

From $U = \llbracket\tau'\rrbracket^+$ we get that either $U = \texttt{Object}$ (then $\tau' = \textsf{Top}$ and we are done) or that

$$\begin{aligned}
\tau' &= \forall\alpha_0{\leq}\tau_0^- \ldots \alpha_n{\leq}\tau_n^-\,.\,\neg\,\sigma' \\
U &= \neg\,\overbrace{\begin{array}{l}\exists X^{\alpha_0} \ldots X^{\alpha_n}\,Y\,\texttt{where}\,X^{\alpha_0}\,\texttt{extends}\,\llbracket\tau_0\rrbracket^- \ldots \\ \qquad\qquad\qquad X^{\alpha_n}\,\texttt{extends}\,\llbracket\tau_n\rrbracket^- \\ \qquad\qquad\qquad Y\,\texttt{extends}\,\llbracket\sigma'\rrbracket^- \\ \qquad\qquad\qquad .\,C^{n+2}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>}\end{array}}^{=U'} \\
&= \exists X\,\texttt{where}\,X\,\texttt{super}\,U'\,.\,D^{1}\texttt{<}X\texttt{>}
\end{aligned}$$

From $\Delta \vdash' T \leq U$ we get by inverting the rules:

$$\textsc{s}_2\text{-OPEN'}\ \dfrac{\begin{array}{c}\textsc{s}_2\text{-ABSTRACT'}\ \dfrac{\textsc{e}_2\text{-SUPER'}\ \dfrac{\mathcal{D} :: \Delta, X\,\texttt{super}\,T' \vdash' U' \leq X}{\Delta, X\,\texttt{super}\,T' \Vdash' X\,\texttt{super}\,U'}}{\Delta, X\,\texttt{super}\,T' \vdash' D^{1}\texttt{<}X\texttt{>} \leq \exists X\,\texttt{where}\,X\,\texttt{super}\,U'\,.\,D^{1}\texttt{<}X\texttt{>}} \\ X \notin \textsf{ftv}(\Delta, U)\end{array}}{\Delta \vdash' \exists X\,\texttt{where}\,X\,\texttt{super}\,T'\,.\,D^{1}\texttt{<}X\texttt{>} \leq \exists X\,\texttt{where}\,X\,\texttt{super}\,U'\,.\,D^{1}\texttt{<}X\texttt{>}}$$

We have $X \notin \mathsf{ftv}(\Delta, T', U')$ so with Lemma 8

$$\begin{gathered}\mathcal{D}' :: \Delta \vdash' U' \leq T' \\ \mathsf{height}(\mathcal{D}') \leq \mathsf{height}(\mathcal{D})\end{gathered}$$

$\mathcal{D}'$ must end with rule $\textsc{s}_2\textsc{-open}$'. Define

$$\begin{gathered}\Delta' = \Delta, X^{\alpha_0}\ \texttt{extends}\ [\![\tau_0]\!]^-, \ldots, X^{\alpha_n}\ \texttt{extends}\ [\![\tau_n]\!]^- \\ \Delta'' = \Delta', Y\ \texttt{extends}\ [\![\sigma']\!]^-\end{gathered}$$

Inverting the rules yields

$$\textsc{s}_2\textsc{-open'}\ \dfrac{\textsc{s}_2\textsc{-abstract'}\ \dfrac{\ldots \quad \textsc{e}_2\textsc{-extends}\ \dfrac{\mathcal{D}'' :: \Delta'' \vdash' Y \leq [\![\sigma]\!]^+}{\Delta'' \Vdash' Y\ \texttt{extends}\ [\![\sigma]\!]^+} \quad \ldots}{\Delta'' \vdash'\ C^{n+2}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>} \leq T'}}{\mathcal{D}' :: \Delta \vdash' U' \leq T'}$$

We have $Y \notin \mathsf{ftv}(\Delta', [\![\sigma']\!]^-, [\![\sigma]\!]^+)$. Hence with Lemma 8

$$\begin{gathered}\mathcal{D}''' :: \Delta' \vdash' [\![\sigma']\!]^- \leq [\![\sigma]\!]^+ \\ \mathsf{height}(\mathcal{D}''') \leq \mathsf{height}(\mathcal{D}'')\end{gathered}$$

Because $\mathcal{D}'''$ is smaller than the initial derivation, we can apply the I.H. and get

$$\Gamma, \alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n \vdash \sigma' \leq \sigma$$

With rule $\textsc{d-all-neg}$

$$\Gamma \vdash \forall\alpha_0 \ldots \alpha_n \,.\, \neg\,\sigma \leq \forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n \,.\, \neg\,\sigma'$$

as required.

– *Case* rule $\textsc{s}_2\textsc{-abstract}$': Impossible because no class type N is in the image of the $[\![\cdot]\!]^-$ translation.

End case distinction on the last rule of this derivation. □

Our final goal is to prove that subtyping in $\mathcal{EX}_{uplo}$, restricted to the image of the translation defined in Fig. 4, coincides with subtyping in $F^{\mathcal{D}}_{\leq}$. We first prove a standard weakening lemma.

Lemma 11 (Weakening). *If $\Delta \vdash T \leq U$ and $\Delta \subseteq \Delta'$ then $\Delta' \vdash T \leq U$.*

Proof. Straightforward induction on the derivation given. □

The next lemma show that the negation operator for types (defined in Fig. 4) allows us to swap the left- and right-hand sides of a subtyping judgment.

Lemma 12. *If $\Delta \vdash U \leq T$ then $\Delta \vdash \neg\,T \leq \neg\,U$.*

Proof. We have

$$\begin{aligned} \neg\, T &= \exists X\ \texttt{where}\ X\ \texttt{super}\ T \,.\, D^{\mathit{1}}\texttt{<}X\texttt{>} \\ \neg\, U &= \exists X\ \texttt{where}\ X\ \texttt{super}\ U \,.\, D^{\mathit{1}}\texttt{<}X\texttt{>} \end{aligned}$$

Assume $\Delta \vdash U \le T$. Then $\Delta, X\ \texttt{super}\ T \vdash U \le T$ with Lemma 11. Hence

$$\text{S}_2\text{-OPEN}\ \dfrac{\begin{array}{c}\text{S}_2\text{-ABSTRACT}\ \dfrac{\text{E}_2\text{-SUPER}\ \dfrac{\text{S}_2\text{-SUPER}\ \dfrac{\Delta, X\ \texttt{super}\ T \vdash U \le T}{\Delta, X\ \texttt{super}\ T \vdash U \le X}}{\Delta, X\ \texttt{super}\ T \Vdash X\ \texttt{super}\ U}}{\Delta, X\ \texttt{super}\ T \vdash D^{\mathit{1}}\texttt{<}X\texttt{>} \le \exists X\ \texttt{where}\ X\ \texttt{super}\ U \,.\, D^{\mathit{1}}\texttt{<}X\texttt{>}} \\ X \notin \mathsf{ftv}(\Delta, \neg\, U)\end{array}}{\Delta \vdash \exists X\ \texttt{where}\ X\ \texttt{super}\ T \,.\, D^{\mathit{1}}\texttt{<}X\texttt{>} \le \exists X\ \texttt{where}\ X\ \texttt{super}\ U \,.\, D^{\mathit{1}}\texttt{<}X\texttt{>}}$$

□

The relation $\Delta \vdash \overline{P} \precsim \overline{Q}$, defined next, expresses that the constraints $\overline{P}$ are more specific than the constraints $\overline{Q}$.

Definition 2 **(**$\Delta \vdash \overline{P} \precsim \overline{P}$**,** $\Delta \vdash P \precsim P$**).**

$$\dfrac{(\forall i \in [n], \exists j \in [m])\ \Delta, \Delta_i \vdash P_j \precsim Q_i\ \textit{with}\ \Delta_i \subseteq \overline{P}}{\Delta \vdash \overline{P}^{m} \precsim \overline{Q}^{n}}$$

$$\dfrac{\Delta \vdash T \le T'}{\Delta \vdash X\ \texttt{extends}\ T \precsim X\ \texttt{extends}\ T'} \qquad \dfrac{\Delta \vdash T' \le T}{\Delta \vdash X\ \texttt{super}\ T \precsim X\ \texttt{super}\ T'}$$

We now connect $\precsim$ with subtyping on existentials.

Lemma 13. *If* $\Delta \vdash \overline{P} \precsim \overline{Q}$ *then* $\Delta \vdash \exists \overline{X}\ \texttt{where}\ \overline{P} \,.\, N \le \exists \overline{X}\ \texttt{where}\ \overline{Q} \,.\, N$.

Proof. It is easy to see that $\Delta \vdash \overline{P} \precsim \overline{Q}$ implies $\Delta, \overline{P} \Vdash Q$ for all $Q \in \overline{Q}$. Then we have

$$\text{S}_2\text{-OPEN}\ \dfrac{\begin{array}{c}\text{S}_2\text{-ABSTRACT}\ \dfrac{(\forall i)\ \Delta, \overline{P} \Vdash Q_i}{\Delta, \overline{P} \vdash N \le \exists \overline{X}\ \texttt{where}\ \overline{Q} \,.\, N} \\ \overline{X} \cap \mathsf{ftv}(\Delta, \exists \overline{X}\ \texttt{where}\ \overline{Q} \,.\, N) = \emptyset\end{array}}{\Delta \vdash \exists \overline{X}\ \texttt{where}\ \overline{P} \,.\, N \le \exists \overline{X}\ \texttt{where}\ \overline{Q} \,.\, N}$$

□

To finish the proof of Theorem 2, we show that subtyping in $\mathcal{EX}_{uplo}$, restricted to the image of the translation defined in Fig. 4, coincides with subtyping in $F^{D}_{\le}$.

Lemma 14. $\Gamma \vdash \tau \le \tau'$ *if and only if* $[\![\Gamma \vdash \tau \le \tau']\!]$.

Proof.

- Assume $\Gamma \vdash \tau \le \tau'$. We proceed by induction on this derivation
  Case distinction on the last rule used.

- *Case* rule d-top: Then $\llbracket\tau'\rrbracket^+ = \texttt{Object}$ and the claim is obvious.
- *Case* rule d-var: Then $\tau = \alpha$ and

  $$\frac{\mathit{\Gamma} \vdash \mathit{\Gamma}(\alpha) \leq \tau' \qquad \tau' \neq \mathsf{Top}}{\mathit{\Gamma} \vdash \alpha \leq \tau'}$$

  Then

  $$X^{\alpha}\,\texttt{extends}\,\llbracket\mathit{\Gamma}(\alpha)\rrbracket^- \in \llbracket\mathit{\Gamma}\rrbracket^-$$

  and by the I.H.

  $$\llbracket\mathit{\Gamma}\rrbracket^- \vdash \llbracket\mathit{\Gamma}(\alpha)\rrbracket^- \leq \llbracket\tau'\rrbracket^+$$

  The claim now follows with rules s2-extends and s2-trans.
- *Case* rule d-all-neg: Then

  $$\frac{\mathit{\Gamma}, \alpha_0 \leq \tau_0 \ldots \alpha_n \leq \tau_n \vdash \sigma' \leq \sigma}{\mathit{\Gamma} \vdash \underbrace{\forall\alpha_0 \ldots \alpha_n \,.\, \neg\,\sigma}_{=\tau} \leq \underbrace{\forall\alpha_0 \leq \tau_0 \ldots \alpha_n \leq \tau_n \,.\, \neg\,\sigma'}_{=\tau'}}$$

  and

  $$\begin{aligned}
  \llbracket\tau\rrbracket^- &= \neg\;\overbrace{\begin{array}{l}\exists X^{\alpha_0} \ldots X^{\alpha_n}\,Y\,\texttt{where}\,Y\,\texttt{extends}\,\llbracket\sigma\rrbracket^+\\ \qquad\qquad\qquad\qquad .\; C^{n+2}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>}\end{array}}^{=T}\\
  \llbracket\tau'\rrbracket^+ &= \neg\;\overbrace{\begin{array}{l}\exists X^{\alpha_0} \ldots X^{\alpha_n}\,Y\,\texttt{where}\,X^{\alpha_0}\,\texttt{extends}\,\llbracket\tau_0\rrbracket^- \ldots\\ \qquad\qquad\qquad\qquad\; X^{\alpha_n}\,\texttt{extends}\,\llbracket\tau_n\rrbracket^-\\ \qquad\qquad\qquad\qquad\; Y\,\texttt{extends}\,\llbracket\sigma'\rrbracket^-\\ \qquad\qquad\qquad\qquad .\; C^{n+2}\texttt{<}Y, X^{\alpha_0} \ldots X^{\alpha_n}\texttt{>}\end{array}}^{=U}
  \end{aligned}$$

  Define

  $$\begin{aligned}
  \mathit{\Delta} &= \llbracket\mathit{\Gamma}\rrbracket^-\\
  \mathit{\Delta}' &= \mathit{\Delta}, X^{\alpha_0}\,\texttt{extends}\,\llbracket\tau_0\rrbracket^- \ldots X^{\alpha_n}\,\texttt{extends}\,\llbracket\tau_n\rrbracket^-
  \end{aligned}$$

  Note that $\llbracket\mathit{\Gamma}, \alpha_0 \leq \tau_0 \ldots \alpha_n \leq \tau_n\rrbracket^- = \mathit{\Delta}'$.
  We must show $\mathit{\Delta} \vdash \neg\,T \leq \neg\,U$. By applying the I.H. we get

  $$\mathit{\Delta}' \vdash \llbracket\sigma'\rrbracket^- \leq \llbracket\sigma\rrbracket^+$$

  Thus

  $$\mathit{\Delta}' \vdash Y\,\texttt{extends}\,\llbracket\sigma'\rrbracket^- \precsim Y\,\texttt{extends}\,\llbracket\sigma\rrbracket^+$$

  Hence

  $$\begin{aligned}
  \mathit{\Delta} \vdash\; & X^{\alpha_0}\,\texttt{extends}\,\llbracket\tau_0\rrbracket^- \ldots X^{\alpha_n}\,\texttt{extends}\,\llbracket\tau_n\rrbracket^-, Y\,\texttt{extends}\,\llbracket\sigma'\rrbracket^-\\
  & \precsim Y\,\texttt{extends}\,\llbracket\sigma\rrbracket^+
  \end{aligned}$$

By Lemma 13

$$\Delta \vdash U \leq T$$

By Lemma 12

$$\Delta \vdash \neg T \leq \neg U$$

End case distinction on the last rule used.

– Assume $[\![\Gamma \vdash \tau \leq \tau']\!]$. Let

$$\begin{aligned}\Delta &= [\![\Gamma]\!]^- \\ T &= [\![\tau]\!]^- \\ U &= [\![\tau']\!]^+\end{aligned}$$

Hence, $\Delta \vdash T \leq U$. By Lemma 6 we then have $\Delta \vdash' T \leq U$. Thus, by Lemma 10, $\Gamma \vdash \tau \leq \tau'$. □

C Example for Loss of Minimal Types with Implementation Definitions for Interface Types

In this section, we demonstrate that implementation definitions for interface types lead to a loss of minimal types. We do not formalize the typing rules for expressions but argue on an informal level. The example relies on explicit implementing types [21], a feature of JavaGI not discussed in this article.

Assume the following definitions:

```
interface I [X] { X m(); }
interface J1
interface J2
class C
class D
class E
implementation J1 [C]
implementation J2 [C]
implementation J1 [D]
implementation J2 [E]
implementation I [J1] { /* illegal */
  J1 m() {
    return new D();
  }
}
implementation I [J2] { /* illegal */
  J2 m() {
    return new E();
  }
}
```

Class `C` is a subtype of `J1` and `J2` because it implements both interfaces. Moreover, the constraints `J1 implements I` and `J2 implements I` holds. Now look at the expression

$e =$ **`new`** `C().m()`

The constraint `C` **`implements`** `I` does not hold. Otherwise, e could be given the type `C`, but evaluating e either produces a `D`- or an `E`-object, both of which are incompatible with `C`. Hence, e cannot have type `C`.

By use of subsumption, we can give **`new`** `C()` the types `J1` and `J2`. Hence, e could be given the types `J1` or `J2`. These two types are incomparable. Their greatest lower bound is `C`, but e cannot have type `C`, so minimal types do not exist.