Subtyping Existential Types

Stefan Wehr and Peter Thiemann

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

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 an accompanying technical report [22].

[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 $\exists X \,\texttt{where}\, \overline{P} \,.\, X$. 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 ".".

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

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

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

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

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 $\{(u_1, v_1) \ldots, (u_n, v_n)\}$ 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 $i_1 \ldots i_r$ such that $u_{i_1} \ldots u_{i_r} = v_{i_1} \ldots v_{i_r}$. The decision problem asks whether such a solution exists.*

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

Proof. Let $\mathcal{P} = \{(u_1, v_1), \ldots, (u_n, v_n)\}$ 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 $i_1, \ldots, i_k$ such that $u = u_{i_1} \ldots u_{i_k}$ and $v = v_{i_1} \ldots v_{i_k}$. To model valid transitions between search states, we define implementations of `S` for all $i \in \{1, \ldots, n\}$ 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:

```
implementation<X> G [∃ S<X,X>]
```
(2)

To get the search running we ask whether there exists some $i \in \{1, \ldots, n\}$ 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 our technical report [22], we prove that the given PCP instance $\mathcal{P}$ has a solution if and only if there exists some $i \in \{1, \ldots, n\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash \exists \texttt{S<}[\![u_i]\!], [\![v_i]\!]\texttt{>} \leq \exists \texttt{G}$ is derivable. □

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

$$\text{E}_2\text{-EXTENDS}\ \frac{\Delta \vdash T \leq U}{\Delta \Vdash T\ \texttt{extends}\ U} \qquad \text{E}_2\text{-SUPER}\ \frac{\Delta \vdash U \leq T}{\Delta \Vdash T\ \texttt{super}\ U}$$

$$\text{S}_2\text{-REFL}\ \Delta \vdash T \leq T \qquad \text{S}_2\text{-TRANS}\ \frac{\Delta \vdash T \leq U \qquad \Delta \vdash U \leq V}{\Delta \vdash T \leq V}$$

$$\text{S}_2\text{-OBJECT}\ \Delta \vdash T \leq \texttt{Object} \qquad \text{S}_2\text{-EXTENDS}\ \frac{X\ \texttt{extends}\ T \in \Delta}{\Delta \vdash X \leq T} \qquad \text{S}_2\text{-SUPER}\ \frac{X\ \texttt{super}\ T \in \Delta}{\Delta \vdash T \leq X}$$

$$\text{S}_2\text{-OPEN}\ \frac{\Delta, \overline{P} \vdash N \leq T \qquad \overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset}{\Delta \vdash \exists \overline{X}\ \texttt{where}\ \overline{P}\,.\,N \leq T} \qquad \text{S}_2\text{-ABSTRACT}\ \frac{T = [\overline{U/X}]N \qquad (\forall i)\ \Delta \Vdash [\overline{U/X}]P_i}{\Delta \vdash T \leq \exists \overline{X}\ \texttt{where}\ \overline{P}\,.\,N}$$

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 S$_2$-EXTENDS and S$_2$-SUPER use assumptions from Δ.

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

$$\tau^+ ::= \mathsf{Top} \mid \forall\alpha_0{\leq}\tau_0^- \ldots \alpha_n{\leq}\tau_n^- \,.\, \neg\, \tau^- \qquad (n \in \mathbb{N})$$
$$\tau^- ::= \alpha \mid \forall\alpha_0 \ldots \alpha_n \,.\, \neg\, \tau^+ \qquad (n \in \mathbb{N})$$
$$\Gamma^- ::= \emptyset \mid \Gamma^-, \alpha{\leq}\tau^-$$

D-TOP
$$\Gamma \vdash \tau \leq \mathsf{Top}$$

D-VAR
$$\frac{\tau \neq \mathsf{Top} \qquad \Gamma \vdash \Gamma(\alpha) \leq \tau}{\Gamma \vdash \alpha \leq \tau}$$

D-ALL-NEG
$$\frac{\Gamma, \alpha_0{\leq}\phi_0 \ldots \alpha_n{\leq}\phi_n \vdash \tau \leq \sigma}{\Gamma \vdash \forall\alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma \leq \forall\alpha_0{\leq}\phi_0 \ldots \alpha_n{\leq}\phi_n \,.\, \neg\, \tau}$$

Fig. 3. Syntax and Subtyping for $F^D_{\leq}$

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

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

Fig. 3 defines the syntax and the subtyping relation of $F^D_{\leq}$. 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 $F^D_{\leq}$ has the form $\Gamma^- \vdash \sigma^- \leq \tau^+$, 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 $\leq$ symbol. The subtyping rule D-ALL-NEG compares two quantified types $\sigma = \forall\alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma'$ and $\tau = \forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\alpha_n \,.\, \neg\, \tau'$ by swapping the left- and right-hand sides of the subtyping judgment and checking $\tau' \leq \sigma'$ under the extended environment $\Gamma, \alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n$. The rule is correct with respect to $F_{\leq}$ because we may interpret every $F^D_{\leq}$ type as an $F_{\leq}$ type:

$$\forall\alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma' = \forall\alpha_0{\leq}\mathsf{Top} \ldots \forall\alpha_n{\leq}\mathsf{Top}.\forall\beta{\leq}\sigma' \,.\, \beta \;(\beta \text{ fresh})$$
$$\forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\alpha_n \,.\, \neg\, \tau' = \forall\alpha_0{\leq}\tau_0 \ldots \forall\alpha_n{\leq}\tau_n.\forall\beta{\leq}\tau' \,.\, \beta \qquad (\beta \text{ fresh})$$

Using these abbreviations, every $F^D_{\leq}$ subtyping judgment can be read as an $F_{\leq}$ subtyping judgment. The subtype relations in $F^D_{\leq}$ and $F_{\leq}$ 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 $\mathsf{dom}(\alpha_1{\leq}\tau_1, \ldots, \alpha_n{\leq}\tau_n) = \{\alpha_1, \ldots, \alpha_n\}$) and, if $\tau = \forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n \,.\, \neg\, \sigma$, then no α_i appears free in any τ_j. A type environment Γ is closed if $\Gamma = \emptyset$ or $\Gamma = \Gamma', \alpha{\leq}\tau$ with Γ' closed and τ closed under Γ'. A judgment $\Gamma \vdash \tau \leq \sigma$ 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 $F^D_{\leq}$. Fig. 4 defines a translation from $F^D_{\leq}$ types, type environments, and subtyping judgments to their corresponding $\mathcal{EX}_{uplo}$ forms. The translation of an n-ary subtyping judgment assumes the

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

Fig. 4. Reduction from $F^D_{\leq}$ 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 $[\![\cdot]\!]^+$ and $[\![\cdot]\!]^-$ indicate whether the translation acts on positive or negative entities.

An n-positive type $\forall \alpha_0 \leq \tau_0^- \ldots \alpha_n \leq \tau_n^- \,.\, \neg\, \tau^-$ 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 $[\![\tau]\!]^-$ of the fresh type variable Y represents the body $\neg\, \tau^-$ of the original type. We cannot use $[\![\tau]\!]^-$ 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 $F^D_{\leq}$ subtyping rule D-ALL-NEG, which swaps the left- and right-hand sides of subtyping judgments.

We now need to verify that $\Gamma \vdash \tau \leq \sigma$ is derivable in $F^D_{\leq}$ if and only if $[\![\Gamma \vdash \tau \leq \sigma]\!]$ is derivable in $\mathcal{EX}_{uplo}$. The "$\Rightarrow$" direction is an easy induction on the derivation of $\Gamma \vdash \tau \leq \sigma$. The "$\Leftarrow$" 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 $[\![\Gamma \vdash \tau \leq \sigma]\!]$ 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 the technical report [22] 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 the technical report [22] 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`.