On the Decidability of Subtyping with Bounded Existential Types[⋆]

Stefan Wehr and Peter Thiemann

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

Abstract. Bounded existential types are a powerful language feature for modeling partial data abstraction and information hiding. However, existentials do not mingle well with subtyping as found in current object-oriented languages: the subtyping relation is already undecidable for very restrictive settings.
This paper considers two subtyping relations defined by extracting the features specific to existentials from current language proposals (JavaGI, WildFJ, and Scala) and shows that both subtyping relations are undecidable. One of the two subtyping relations remains undecidable even if bounded existential types are removed.
With the goal of regaining decidable type checking for the JavaGI language, the paper also discusses various restrictions including the elimination of bounded existentials from the language as well as possible amendments to regain some of their features.

1 Introduction

Cardelli and Wegner [5] introduced bounded existential types to obtain a fine-grained modeling instrument for structured and partial data abstraction and information hiding, thus generalizing the concept of an abstract data type modeled with a plain existential type [12]. Language designers and type theorists rely on bounded existentials in diverse areas such as object-oriented languages [1], module systems [19], and functional languages [10].

In the realm of object-oriented languages, bounded existential types have found uses for modeling object-oriented languages in general [2], as well as for modeling specific features such as Java wildcards [3,20,21]. Only a few languages (e.g., Scala [13]) make bounded existential types in full generality available to the programmer. Also, the initial design of JavaGI [23] includes bounded existential types and provides interface types (i.e., the ability to form types from interface names) as a special case supported by syntactic sugar. Building directly on bounded existential types has several advantages compared to interface types: they properly generalize interface types, they encompass Java wildcards, and they have meaningful uses with interfaces abstracting over families of types. Thus, despite the complexity that they introduce in a type system, bounded existential types initially appear like a worthwhile feature.

[⋆] A preliminary version of this work was presented at FTfJP 2008 [24].

Unfortunately, it turns out that subtyping —and hence type checking— for JavaGI is undecidable in the presence of general bounded existential types. Furthermore, subtyping for bounded existential types as used to encode Java wildcards is also undecidable. This article proves both of these undecidability results. Moreover, it also shows that replacing JavaGI's existentials with plain interface types does not regain decidability of subtyping.

Contributions and Overview

After refreshing some background on the JavaGI language in Sec. 2, Sec. 3 defines the calculus $\mathcal{EX}_{impl}$ that models the essential aspects of subtyping and bounded existential types in JavaGI. Subtyping in $\mathcal{EX}_{impl}$ is shown to be undecidable by reduction from Post's Correspondence Problem [18]. Further, the section defines the calculus $\mathcal{IT}_{impl}$ by replacing existential types in $\mathcal{EX}_{impl}$ with plain interface types. Subtyping in $\mathcal{IT}_{impl}$ is also undecidable but various restrictions exist that ensure decidability.

Sec. 4 considers the calculus $\mathcal{EX}_{uplo}$ supporting existentials with lower and upper bounds. Subtyping in $\mathcal{EX}_{uplo}$ is also undecidable, as shown by reduction from subtyping in $F^D_\leq$, a restricted form of the polymorphic λ-calculus extended with subtyping [14]. The results in this section are relevant to Scala [13], formal systems for modeling Java wildcards [3,4,20], and JavaGI's full type system.

Sec. 5 explores alternative design options for JavaGI that avoid bounded existential types but keep the remaining features. Finally, Sec. 6 reviews related work and Sec. 7 concludes. Detailed proofs may be found in an accompanying technical report [26].

2 Background

JavaGI [23,25] is a conservative extension of Java 1.5. It generalizes Java's interface concept to incorporate the essential features of Haskell type classes [8,22]. The generalization allows for retroactive and type-conditional interface implementations, binary methods, static methods in interfaces, default implementations for interface methods, and multi-headed interfaces (interfaces over families of types). Furthermore, JavaGI's initial design generalizes Java-like interface types to bounded existential types. This section only discusses the features relevant to this paper, namely retroactive interface implementations and existential types.

2.1 Retroactive Interface Implementations

Retroactive interface implementations allow programmers to implement an interface for some class without changing the source code of the class. 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`:[1]

[1] 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()`.

```
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++); }
    };
  }
}
```

Fig. 1. Retroactive implementation of the `Iterable` interface.

```
for (Character c : "21 is only half the truth") { ... } // illegal in Java
```

As a class definition in Java must specify all interfaces that the class implements and the definition of `java.lang.String` is fixed, there is no hope of getting this code to work. A JavaGI programmer can overcome this restriction by adding implementations for interfaces to an existing class at any time, retroactively, without modifying the source code of the class.

For example, the *implementation definition* shown in Fig. 1 specifies that the *implementing type* `String`, enclosed in square brackets `[]`, implements the interface `Iterable<Character>`.[2] The definition of the `iterator` method can use the methods `length` and `charAt` because they are part of `String`'s public interface.

2.2 Bounded Existential Types

Java uses the name of an interface as an *interface type* to denote the set of all types implementing the interface. Instead of interface types, the initial design of JavaGI features *bounded existential types* [5] 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 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. (The occurrence of "`List<String>`" in the implementation constraint does not abbreviate an existential 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, 20, 21]. For instance, the existential type ∃ `X` **`where`** `X` **`extends`** `Number . List<X>` corresponds to the wildcard type `List<?` **`extends`** `Number>`.[3]

[2] The implementation ignores the `remove` method of the `Iterator` interface.

[3] 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`.

$\boxed{\text{Syntax}}$

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

$$X, Y, Z \in \mathit{TyvarName} \qquad I, J \in \mathit{IfaceName}$$

$\boxed{\Theta; \Delta \Vdash T\ \texttt{implements}\ I\texttt{<}\overline{T}\texttt{>}}$

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

$\boxed{\Theta; \Delta \vdash T \leq T}$

$$
\text{S}_1\text{-REFL}\ \ \Theta; \Delta \vdash T \leq T
\qquad
\text{S}_1\text{-TRANS}\ \frac{\Theta; \Delta \vdash T \leq U \qquad \Theta; \Delta \vdash U \leq V}{\Theta; \Delta \vdash T \leq V}
$$

$$
\text{S}_1\text{-OPEN}\ \frac{\Theta; \Delta, P \vdash X \leq T \qquad X \notin \mathsf{ftv}(\Theta, \Delta, T)}{\Theta; \Delta \vdash (\exists X\ \texttt{where}\ P\,.\,X) \leq T}
\qquad
\text{S}_1\text{-ABSTRACT}\ \frac{\Theta; \Delta \Vdash [T/X]P}{\Theta; \Delta \vdash T \leq (\exists X\ \texttt{where}\ P\,.\,X)}
$$

Fig. 2. Syntax, entailment, and subtyping for $\mathcal{EX}_{\mathit{impl}}$.

The initial design of JavaGI allows implementation definitions for existentials. For example, given an interface `I`, a programmer may write an implementation definition to specify that all types implementing `List<X>` also implement `I`.

```
implementation<X> I [List<X>] { /* implement methods of I */ }
```

Such a definition is feasible only if all methods of `I` can be implemented using only methods of the `List` interface. The example also demonstrates that JavaGI supports *generic implementation definitions*, which are parameterized by type variables.

3 Subtyping Existential Types with Implementation Constraints

This section introduces $\mathcal{EX}_{\mathit{impl}}$, a subtyping calculus with existentials ($\mathcal{EX}$) and implementation constraints (*impl*). The calculus is a subset of Core-JavaGI [23]. It does not model all aspects of JavaGI's initial design, but contains only those features that make subtyping undecidable. In particular, the syntax of existentials is restricted such that all existentials are encodings of interface types. Consequently, undecidability of subtyping also holds for $\mathcal{IT}_{\mathit{impl}}$, a variant of $\mathcal{EX}_{\mathit{impl}}$ where interface types ($\mathcal{IT}$) replace existentials.

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

Fig. 2 defines the syntax along with the entailment and subtyping relations of $\mathcal{EX}_{\mathit{impl}}$. 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`<`$\overline{T}$`>`. An interface without type parameters is written I instead of I`<`$\bullet$`>`.[4]

A type T is either a type variable X or a bounded existential type of the form $\exists X$ `where` X `implements` I`<`$\overline{T}$`>` $. X$. For simplicity, there are no class types, existentials have a single quantified type variable X, they have exactly one constraint X `implements` I`<`$\overline{T}$`>`, and the body of an existential must be the quantified type variable. Existentials are considered equal up to renaming of bound type variables.

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 methods do not matter for the entailment and subtyping relations of $\mathcal{EX}_{impl}$. Moreover, $\mathcal{EX}_{impl}$ does not support interface inheritance. An implementation definition `implementation<`$\overline{X}$`>` I`<`$\overline{T}$`>` `[`J`<`$\overline{U}$`>]` implicitly assumes that $\overline{X} = \mathsf{ftv}(J$`<`$\overline{U}$`>`$)$.[5]

The entailment relation $\Theta; \Delta \Vdash T$ `implements` I`<`$\overline{T}$`>` expresses that type T implements interface I`<`$\overline{T}$`>`. It relies on a program environment Θ, which is a finite set of definitions *def*, and a type environment Δ, which is a finite set of constraints P, where Δ, P abbreviates $\Delta \cup \{P\}$. A type implements an interface either because it corresponds to an instance of a suitable implementation definition (rule E₁-IMPL) or because the type environment contains the constraint (rule E₁-LOCAL).[6]

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 S₁-OPEN opens an existential on the left-hand side of the subtyping relation by moving its constraint 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 S₁-ABSTRACT deals with existentials on the right-hand side of the subtyping relation. It states that T is a subtype of some existential if the constraint of the existential holds after substituting T for the existentially quantified type variable.

As part of 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 $\mathcal{EX}_{impl}$

The undecidability of subtyping in $\mathcal{EX}_{impl}$ follows by reduction from Post's Correspondence Problem (PCP). It is well known that PCP is undecidable [7, 18].

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*

[4] The notation $\overline{\xi}$ abbreviates a sequence $\xi_1, \ldots, \xi_n$ of syntactic entities with $\bullet$ standing for the empty sequence. Sometimes, the sequence $\overline{\xi}$ stands for the set $\{\overline{\xi}\}$.

[5] The notation $\mathsf{ftv}(\xi)$ denotes the set of type variables free in ξ.

[6] The notation $[\overline{T/X}]$ stands for the capture-avoiding substitution replacing each X_i with T_i.

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 2. *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

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

The abbreviation $I\texttt{<}\overline{T}\texttt{>}$ 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 `A<B<E>>`.

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

$$\varepsilon \mathbin{\#} T := T \qquad\qquad Lu \mathbin{\#} T := L\texttt{<}u \mathbin{\#} T\texttt{>}$$

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 $\texttt{S<}[\![u]\!], [\![v]\!]\texttt{>}$ 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:

```
implementation<X,Y> S<uᵢ#X, vᵢ#Y> [S<X,Y>]                    (1)
```

The type `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 $i \in \{1, \dots, n\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash \texttt{S<}[\![u_i]\!], [\![v_i]\!]\texttt{>} \leq \texttt{G}$ is derivable. The program $\Theta_{\mathcal{P}}$ consists of the interfaces and implementations just defined. In the extended version [26], we prove a lemma showing that $\mathcal{P}$ has a solution if, and only if, there exists some $i \in \{1, \dots, n\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash \texttt{S<}[\![u_i]\!], [\![v_i]\!]\texttt{>} \leq \texttt{G}$ is derivable. □

Example. Suppose the PCP instance $\mathcal{P} = \{(u_1, v_1), (u_2, v_2)\}$ with $u_1 = \texttt{A}$, $u_2 = \texttt{ABA}$, $v_1 = \texttt{AA}$, and $v_2 = \texttt{B}$ is given. The instance has the solution $1, 2, 1$ because $u_1 u_2 u_1 = v_1 v_2 v_1 = \texttt{AABAA}$. The program $\Theta_{\mathcal{P}}$ for the $\mathcal{EX}_{impl}$ encoding of this problem looks like this:

$\boxed{\text{Syntax}}$

$$\varphi ::= \forall \overline{X}.\, I\texttt{<}\overline{T}\texttt{>} \leq I\texttt{<}\overline{T}\texttt{>}$$
$$T, U, V, W ::= X \mid I\texttt{<}\overline{T}\texttt{>}$$
$$X, Y, Z \in \mathit{TyvarName} \qquad I, J \in \mathit{IfaceName}$$

$\boxed{\Phi \vdash T \leq T}$

$$\textsc{s}_2\text{-REFL}\quad \Phi \vdash T \leq T \qquad \textsc{s}_2\text{-TRANS}\quad \frac{\Phi \vdash T \leq U \qquad \Phi \vdash U \leq V}{\Phi \vdash T \leq V} \qquad \textsc{s}_2\text{-IMPL}\quad \frac{(\forall \overline{X}.\, T \leq U) \in \Phi}{\Phi \vdash [\overline{V/X}]T \leq [\overline{V/X}]U}$$

Fig. 3. Syntax and subtyping for $\mathcal{IT}_{impl}$.

```
interface E           interface A<X>     interface B<X>
interface S<X,Y>      interface G
implementation<X,Y> S<A<X>,      A<A<Y>>>  [S<X,Y>]       // (1)
implementation<X,Y> S<A<B<A<X>>>, B<Y>>    [S<X,Y>]       // (2)
implementation<X>   G                      [S<X,X>]       // (3)
```

We then need to ask whether there exists some $i \in \{1, 2\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash$ S<$[\![u_i]\!], [\![v_i]\!]$> $\leq$ G is derivable. Verifying that such a derivation exists for $i = 1$ is left as an exercise to the reader.

3.3 Undecidability Without Existential Types

The proof of undecidability of subtyping in $\mathcal{EX}_{impl}$ reveals that subtyping remains undecidable even if plain interface types replace existentials. To make this claim concrete, Fig. 3 defines the calculus $\mathcal{IT}_{impl}$, which essentially is a version of $\mathcal{EX}_{impl}$ with plain interface types instead of existentials. The simplify the syntax, $\mathcal{IT}_{impl}$ drops interface definitions altogether and uses subtyping schemes instead of implementation definitions: a *subtyping scheme* $\varphi = \forall \overline{X}.\, J\texttt{<}\overline{T}\texttt{>} \leq I\texttt{<}\overline{U}\texttt{>}$ corresponds to an implementation definition implementation<$\overline{X}$> I<$\overline{U}$> [J<$\overline{T}$>]. Such a subtyping scheme implicitly assumes that $\overline{X} = \mathsf{ftv}(J\texttt{<}\overline{T}\texttt{>})$. $\mathcal{IT}_{impl}$ also replaces constraint entailment and the rules $\textsc{s}_1$-OPEN and $\textsc{s}_1$-ABSTRACT with a single rule $\textsc{s}_2$-IMPL. The symbol Φ ranges over finite sets of subtyping schemes φ.

Theorem 3. *Subtyping in $\mathcal{IT}_{impl}$ is undecidable.*

Proof. Similar to the proof of Theorem 2.

The rest of this section investigates decidable fragments of $\mathcal{IT}_{impl}$. It starts with the observation that the undecidability proofs of subtyping in $\mathcal{EX}_{impl}$ and $\mathcal{IT}_{impl}$ rely on two main ingredients:

Cyclic interface subtyping. Implementation definitions in $\mathcal{EX}_{impl}$ (or subtyping schemes in $\mathcal{IT}_{impl}$) allow the introduction of cycles in the subtyping graph of interfaces. Consider one of the implementations defined by Equation (1) on page 6: it states that S<u_i # X, v_i # Y> is a supertype of S<X, Y>. In the reduction from PCP, such cycles are used to encode the individual steps in the search for a solution.

Multiple instantiation subtyping. Implementation definitions in $\mathcal{EX}_{impl}$ (or subtyping schemes in $\mathcal{IT}_{impl}$) allow to introduce two different instantiations of the same interface as supertypes of some other interface. Consider again the implementations defined by Equation (1): for $u_i \neq u_j$ or $v_i \neq v_j$ the implementations state that $\texttt{S<}u_i \# \texttt{X}, v_i \# \texttt{Y>} \neq \texttt{S<}u_j \# \texttt{X}, v_j \# \texttt{Y>}$ are both supertypes of $\texttt{S<X, Y>}$. In the reduction from PCP, multiple instantiation subtyping encodes the choice between different pairs (u_i, v_i) and (u_j, v_j).

An obvious way to obtain decidable subtyping for $\mathcal{IT}_{impl}$ is to restrict the set of subtyping schemes Φ such that, for all types T, only a finite set of T-supertypes is derivable from Φ.

Definition 4. *The* set of T-supertypes derivable from Φ*, written* $\mathcal{S}_{T,\Phi}$*, is defined as the smallest set closed under the following rules:*

$$T \in \mathcal{S}_{T,\Phi} \qquad\qquad \frac{(\forall \overline{X}.\, V \leq U) \in \Phi \qquad [\overline{W/X}]V \in \mathcal{S}_{T,\Phi}}{[\overline{W/X}]U \in \mathcal{S}_{T,\Phi}}$$

Restriction 1. The set $\mathcal{S}_{T,\Phi}$ must be finite for all types T.

Theorem 5. *Under Restriction 1, subtyping in* $\mathcal{IT}_{impl}$ *is decidable.*

Proof. See the extended version [26]. □

Here is a restriction that eliminates cyclic interface subtyping.

Definition 6. *A finite set of subtyping schemes* Φ *is contractive if, and only if, there exists no sequence* $\varphi_1, \ldots, \varphi_n \in \Phi$ *such that* $\varphi_i = \forall \overline{X_i}.\, I_i\texttt{<}\overline{T_i}\texttt{>} \leq J_i\texttt{<}\overline{U_i}\texttt{>}$ *for all* $i = 1, \ldots, n$ *and* $J_i = I_{i+1}$ *for all* $i = 1, \ldots, n-1$ *and* $J_n = I_1$.

Restriction 2. The set Φ must be contractive.

Lemma 7. *Restriction 2 implies Restriction 1.*

Proof. See the extended version [26]. □

Remark. Restriction 1 does not imply Restriction 2. Consider the set $\Phi = \{\forall \bullet .\, I \leq I\}$, which obviously meets Restriction 1 but is not contractive.

The next restriction is strictly stronger than Restriction 2.

Restriction 3. For all $\varphi_1, \varphi_2 \in \Phi$ is must hold that $\varphi_1 = \forall \overline{X}.\, I_1\texttt{<}\overline{T}\texttt{>} \leq J_1\texttt{<}\overline{U}\texttt{>}$ and $\varphi_2 = \forall \overline{Y}.\, I_2\texttt{<}\overline{V}\texttt{>} \leq J_2\texttt{<}\overline{W}\texttt{>}$ imply $J_1 \neq I_2$.

The last restriction considered eliminates multiple instantiation subtyping.

Restriction 4. If $\Phi \vdash I\texttt{<}\overline{T}\texttt{>} \leq J\texttt{<}\overline{U}\texttt{>}$ and $\Phi \vdash I\texttt{<}\overline{T}\texttt{>} \leq J\texttt{<}\overline{V}\texttt{>}$ then it must hold that $\overline{U} = \overline{V}$.

Lemma 8. *Restriction 4 implies Restriction 1.*

Proof. See the extended version [26]. □

Remark. Neither Restriction 1 nor Restriction 2 implies Restriction 4 as demonstrated by $\Phi = \{\forall \bullet .\, I \leq J\texttt{<}A\texttt{>}, \forall \bullet .\, I \leq J\texttt{<}B\texttt{>}\}$. Moreover, Restriction 4 does not imply Restriction 2 as demonstrated by $\Phi' = \{\forall \bullet .\, I \leq J, \forall \bullet .\, J \leq I\}$.

$\boxed{\text{Syntax}}$

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

$$X, Y, Z \in \mathit{TyvarName} \qquad C, D \in \mathit{ClassName}$$

$\boxed{\Delta \Vdash T\,\texttt{extends}\,T \qquad \Delta \Vdash T\,\texttt{super}\,T}$

$$\text{E3-EXTENDS}\ \frac{\Delta \vdash T \leq U}{\Delta \Vdash T\,\texttt{extends}\,U} \qquad \text{E3-SUPER}\ \frac{\Delta \vdash U \leq T}{\Delta \Vdash T\,\texttt{super}\,U}$$

$\boxed{\Delta \vdash T \leq T}$

$$\text{S3-REFL}\ \Delta \vdash T \leq T \qquad \text{S3-TRANS}\ \frac{\Delta \vdash T \leq U \quad \Delta \vdash U \leq V}{\Delta \vdash T \leq V} \qquad \text{S3-OBJECT}\ \Delta \vdash T \leq \texttt{Object} \qquad \text{S3-EXTENDS}\ \frac{X\,\texttt{extends}\,T \in \Delta}{\Delta \vdash X \leq T}$$

$$\text{S3-SUPER}\ \frac{X\,\texttt{super}\,T \in \Delta}{\Delta \vdash T \leq X} \qquad \text{S3-OPEN}\ \frac{\Delta, \overline{P} \vdash N \leq T \quad \overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset}{\Delta \vdash \exists \overline{X}\,\texttt{where}\,\overline{P}\,.\,N \leq T} \qquad \text{S3-ABSTRACT}\ \frac{T = [\overline{U/X}]N \quad (\forall i)\ \Delta \Vdash [\overline{U/X}]P_i}{\Delta \vdash T \leq \exists \overline{X}\,\texttt{where}\,\overline{P}\,.\,N}$$

Fig. 4. 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 (*uplo*) for type variables but no implementation constraints. Other researchers [3,4,20] use formal systems very similar to $\mathcal{EX}_{uplo}$ for modeling Java wildcards [21]. 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. Scala [13] as well as the initial design of the full JavaGI language employ existential types with upper and lower bounds as a replacement for Java wildcards.

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

Fig. 4 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, they support multiple constraints, 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 finite set of constraints P with Δ, P standing for $\Delta \cup \{P\}$.

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 (implicit) restrictions:

$\boxed{\text{Syntax}}$

$$\begin{aligned}
\tau^+ &::= \mathsf{Top} \mid \forall \alpha_0 {\leq} \tau_0^- \dots \alpha_n {\leq} \tau_n^- \,.\, \neg\, \tau^- \\
\tau^- &::= \alpha \mid \forall \alpha_0 \dots \alpha_n \,.\, \neg\, \tau^+ \\
\Gamma^- &::= \emptyset \mid \Gamma^-, \alpha {\leq} \tau^-
\end{aligned}$$

$\boxed{\Gamma^- \vdash \sigma^- \leq \tau^+}$

$$\text{D-TOP} \quad \Gamma \vdash \tau \leq \mathsf{Top} \qquad \text{D-VAR} \quad \frac{\tau \neq \mathsf{Top} \quad \Gamma \vdash \Gamma(\alpha) \leq \tau}{\Gamma \vdash \alpha \leq \tau} \qquad \text{D-ALL-NEG} \quad \frac{\Gamma, \alpha_0 {\leq} \phi_0 \dots \alpha_n {\leq} \phi_n \vdash \tau \leq \sigma}{\Gamma \vdash \forall \alpha_0 \dots \alpha_n \,.\, \neg\, \sigma \leq \forall \alpha_0 {\leq} \phi_0 \dots \alpha_n {\leq} \phi_n \,.\, \neg\, \tau}$$

Fig. 5. Syntax and subtyping for $F_{\leq}^{D}$.

(1) If $T = \exists \overline{X}\ \mathtt{where}\ \overline{P} \,.\, N$, then $\overline{X} \neq \bullet$ and $\overline{X} \subseteq \mathsf{ftv}(N)$. That is, an existential must abstract over at least one type variable and all its bounded type variables must appear in the body type N.
(2) If $T = \exists \overline{X}\ \mathtt{where}\ \overline{P} \,.\, N$ and $P \in \overline{P}$, then $P = Y\ \mathtt{extends}\ T$ or $P = Y\ \mathtt{super}\ T$ 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.[7]

These three restrictions simplify the formulation of a variant of $\mathcal{EX}_{uplo}$'s subtyping relation without an explicit rule for transitivity (see the extended version [26]).

Constraint entailment ($\Delta \Vdash T\ \mathtt{extends}\ U$ and $\Delta \Vdash U\ \mathtt{super}\ T$) 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 S3-EXTENDS and S3-SUPER use assumptions from Δ. Moreover, rule S3-ABSTRACT possibly needs to "guess" some of the types $\overline{U}$ because, unlike in $\mathcal{EX}_{impl}$, existentials in $\mathcal{EX}_{uplo}$ may quantify over more than one type variable.

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

To get a feeling how subtyping derivations in $\mathcal{EX}_{uplo}$ may lead to infinite regress, consider the goal $\{X\ \mathtt{extends}\ \neg\, U\} \vdash X \leq \neg\, C\texttt{<}X\texttt{>}$, where U is defined as $\exists X\ \mathtt{where}\ X\ \mathtt{extends}\ \neg\, C\texttt{<}X\texttt{>} \,.\, C\texttt{<}X\texttt{>}$ and the notation $\neg\, T$ is an abbreviation for $\exists X\ \mathtt{where}\ X\ \mathtt{super}\ T \,.\, D\texttt{<}X\texttt{>}$ such that X is fresh. Searching for a derivation of this goal quickly leads to a subgoal of the form $\{X\ \mathtt{extends}\ \neg\, U, Z\ \mathtt{super}\ U\} \vdash X \leq \neg\, C\texttt{<}X\texttt{>}$, where Z is a fresh type variable introduced by rule S3-OPEN. The details are left as an exercise to the reader.

The undecidability proof of subtyping in $\mathcal{EX}_{uplo}$ is by reduction from $F_{\leq}^{D}$ [14], a restricted version of $F_{\leq}$ [5]. Pierce defines $F_{\leq}^{D}$ for his undecidability proof of $F_{\leq}$ subtyping [14]. Fig. 5 recapitulates $F_{\leq}^{D}$'s syntax and subtyping relation. Let n be a fixed natural number. A type τ is either an n-positive type, τ^+, or an n-negative

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

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

Fig. 6. Reduction from $F_{\leq}^{D}$ to $\mathcal{EX}_{uplo}$.

type, τ^{-}, where n stands for the number of type variables (minus one) bound at the top-level of the type. An 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, the polarity is often omitted and n is left implicit.

An *n-ary subtyping judgment* in $F_{\leq}^{D}$ has the form $\Gamma^{-} \vdash \sigma^{-} \leq \tau^{+}$, where Γ^{-} is an n-negative type environment, σ^{-} is an n-negative type, and τ^{+} is an 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 $F_{\leq}^{D}$ type as an $F_{\leq}$ type:

$$
\begin{aligned}
\forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma' &\equiv \forall \alpha_0 {\leq} \mathsf{Top} \ldots \forall \alpha_n {\leq} \mathsf{Top}. \forall \beta {\leq} \sigma' \,.\, \beta \ (\beta \text{ fresh}) \\
\forall \alpha_0 {\leq} \tau_0 \ldots \alpha_n {\leq} \alpha_n \,.\, \neg\, \tau' &\equiv \forall \alpha_0 {\leq} \tau_0 \ldots \forall \alpha_n {\leq} \tau_n. \forall \beta {\leq} \tau' \,.\, \beta \qquad (\beta \text{ fresh})
\end{aligned}
$$

Using these abbreviations, every $F_{\leq}^{D}$ subtyping judgment can be read as an $F_{\leq}$ subtyping judgment. The subtyping relations in $F_{\leq}^{D}$ 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.

These notions are sufficient to state the central theorem of this section and sketch its proof.

Theorem 9. *Subtyping in $\mathcal{EX}_{uplo}$ is undecidable.*

Proof. The proof is by reduction from $F_{\leq}^{D}$. Fig. 6 defines a translation from $F_{\leq}^{D}$ types, type environments, and subtyping judgments to their corresponding $\mathcal{EX}_{uplo}$ forms. The translation of an n-ary subtyping judgment assumes the 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.

As before, 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.

An n-positive type $\forall\alpha_0{\leq}\tau_0^- \ldots \alpha_n{\leq}\tau_n^-\,.\,\neg\,\tau^-$ is translated into a 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 Sec. 4.1. Type environments and subtyping judgments are translated in the obvious way.

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 S3-TRANS (Fig. 4) 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 extended version [26] for details and the full proof. □

5 Lessons Learned

What are the consequences of this investigation for the design of JavaGI? While bounded existential types are powerful and unify several diverse concepts, they complicate the metatheory of JavaGI's initial design considerably. Also, subtyping with existential types is undecidable in the general case, as demonstrated in the two preceding sections.

There are three alternatives for dealing with this problem: (1) Accept that the subtyping relation is undecidable. (2) Restrict existentials such that subtyping becomes decidable. (3) Remove existentials from JavaGI altogether.

The first alternative (opted for by the Scala compiler) is pragmatic and readily implementable by imposing a resource limit on the subtype checker to avoid divergence. Consequently, the subtyping algorithm would become incomplete with respect to its specification.

The second alternative turns out not to be viable. It is feasible to come up with a set of restrictions that keep the subtyping relation defined in Sec. 3 decidable. (Sec. 3.3 investigated such restrictions for plain interface types.) For the subtyping relation defined in Sec. 4, however, we were not able to identify sensible restrictions without giving up either lower or upper bounds (which are essential for encoding Java wildcards). Moreover, restricting existential types so that subtyping becomes decidable would make an already complex type system even more complicated. In addition, such restrictions tend to be difficult to communicate to users of the language.

The third alternative comes with the realization that existentials may not be worth the trouble. JavaGI's main contribution is its very general and powerful interface concept (which this paper does not explore, but see [23, 25]). While existential types are related to this concept, they are not at the heart of it. In fact, we conducted several real-world case studies using our implementation of JavaGI without a need for full-blown existential types arising [25].

Under these circumstances, it appears that the price of having bounded existential types at the core of JavaGI is too high. Hence, the current, revised version of JavaGI elides bounded existential types because of their poor power/cost ratio but retains all other features of the previous design. Thus, it gives up some of the power in favor of simplicity.

Several already existing features make up for the lack of existentials. More specifically, the revised design copes with most uses of existentials from Sec. 1: parametric polymorphism in combination with multiple subtyping constraints (as already present in Java) allows to emulate the composition of interface types in most situations; direct support for wildcards avoids their encoding through existential types;[8] JavaGI's **`implements`** constraints in combination with parametric polymorphism allow the specification of meaningful types for interfaces over families of types.

Even the revised design has to accept compromises to avoid undecidability. Sec. 3.3 reveals that subtyping remains undecidable even if plain interface types replace existentials. The real culprit for undecidability is the ability to provide implementation definitions with interface types acting as implementing types.

Disallowing such implementation definitions completely is a rather severe restriction because it prevents useful implementation definitions such as the one given for `List<X>` in Sec. 2.2. Instead, the revised design allows interfaces as implementing types but it imposes the equivalent of Restriction 3 from Sec. 3.3: if an interface is used as an implementing type then no retroactive implementation can be provided for this interface.[9]

Sometimes, even this restriction is too strict. For example, the use of `List<X>` as an implementing type in Sec. 2.2 prevents retroactive implementations of the `List` interface. *Abstract implementation definitions* are a potential cure. They look similar to non-abstract implementation definitions but do not contribute to constraint entailment and subtyping, so the restriction just explained does

[8] It is an open question whether subtyping for Java wildcards is decidable (see Sec. 6 for details). Of course, the inclusion of wildcards in JavaGI is a concession to ensure backwards compatibility with Java 1.5. An embedding of JavaGI's generalized interface concept in other languages such as C# could easily drop support for wildcards. Thus, the decidability question for wildcards is not intrinsic to decidability of subtyping in JavaGI.

[9] Restriction 2 is more flexible then Restriction 3 but the latter simplifies the detection of ambiguities arising through conflicting implementation definitions and it allows for an efficient implementation of dynamic method lookup.

not apply. The details of abstract implementation definitions are explained elsewhere [25].

6 Related Work

Kennedy and Pierce [9] 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. The proof in Sec. 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 implementation definitions for existentials (or interface types), which cause cyclic interface and multiple instantiation subtyping.

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 Sec. 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 S3-OPEN and rule S3-ABSTRACT together with lower bounds on type variables play a similar role.

Torgersen and coworkers [20] present WildFJ as a model for Java wildcards using existential types. The authors do not prove WildFJ sound. Cameron and coworkers [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 Sec. 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 S3-OPEN and S3-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 s such that $sT \leq sU$ for Java types T, U with wildcards [16, 17]. Our undecidability result for $\mathcal{EX}_{uplo}$ 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, which is not addressed in this paper. In general, existentials in $\mathcal{EX}_{uplo}$ are strictly more powerful than Java wildcards. For example, the existential $\exists X.C\texttt{<}X, X\texttt{>}$ cannot be encoded as the wildcard type $C\texttt{<?,?>}$ because the two occurrences of `?` denote two distinct types.

The programming language Scala [13] supports existential types in its latest release to provide better interoperability with Java libraries using wildcards and to address the avoidance problem [15, Chapter 8]. The subtyping rules for existentials (Sec. 3.2.10 and Sec. 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.

A recent article [25] reports on practical experience with the revised design of JavaGI. It discusses several case studies and describes the implementation of a compiler and a runtime system for JavaGI, which employs the restrictions discussed in Sec. 5.

7 Conclusion

The paper investigated decidability of subtyping with bounded existential types in the context of JavaGI, Java wildcards, and Scala. It defined two calculi $\mathcal{EX}_{impl}$ and $\mathcal{EX}_{uplo}$ featuring bounded existential types in two variations and proved undecidability of subtyping for both calculi. Subtyping is also undecidable for $\mathcal{IT}_{impl}$, a simplified version of $\mathcal{EX}_{impl}$ without existentials. The paper also suggested a revised version of JavaGI that avoids fully general existentials without giving up much expressivity. The revised version of JavaGI is available at `http://www.informatik.uni-freiburg.de/~wehr/javagi/`.

Acknowledgments

We thank the anonymous FTfJP 2008 and APLAS 2009 reviewers for feedback on earlier versions of this article.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer, 1996.
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 J. Vitek, editor, *22nd ECOOP*, volume 5142 of *LNCS*, Paphos, Cyprus, 2008. Springer.
4. N. Cameron, E. Ernst, and S. Drossopoulou. Towards an existential types model for Java wildcards. In *FTfJP, 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. S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. 2nd ESOP*, volume 300 of *LNCS*, pages 131–144. Springer, 1988.
9. A. J. Kennedy and B. C. Pierce. On decidability of nominal subtyping with variance. In *FOOL/WOOD, informal proceedings*, Jan. 2007. `http://foolwood07.cs.uchicago.edu/program/kennedy-abstract.html`.

10. K. Läufer. Type classes with existential types. *J. Funct. Program.*, 6(3):485–517, May 1996.
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 TOPLAS*, 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. B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.
16. M. Plümicke. Typeless programming in Java 5.0 with wildcards. In *5th PPPJ*. ACM, Sept. 2007.
17. M. Plümicke. Java type unification with wildcards. In *Applications of Declarative Programming and Knowledge Management*, volume 5437 of *LNCS*, pages 223–240. Springer, 2009.
18. E. L. Post. A variant of a recursivley unsolvable problem. *Bulletin of the American Mathematical Society*, 53:264–268, 1946.
19. C. V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, Edinburgh, Scotland, 1998. LFCS Thesis ECS–LFCS–98–389.
20. M. Torgersen, E. Ernst, and C. P. Hansen. Wild FJ. In *FOOL, informal proceedings*, 2005. `http://homepages.inf.ed.ac.uk/wadler/fool/program/14.html`.
21. 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.
22. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th ACM Symp. POPL*, pages 60–76, Austin, Texas, USA, Jan. 1989. ACM Press.
23. S. Wehr, R. Lämmel, and P. Thiemann. JavaGI: Generalized interfaces for Java. In E. Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 347–372, Berlin, Germany, July 2007. Springer.
24. S. Wehr and P. Thiemann. Subtyping existential types. In *10th FTfJP, informal proceedings*, 2008. `http://www.informatik.uni-freiburg.de/~wehr/publications/subex.pdf`.
25. S. Wehr and P. Thiemann. JavaGI in the battlefield: Practical experience with generalized interfaces. In *Proc. 8th GPCE*, Denver, Colorado, USA, 2009. ACM.
26. S. Wehr and P. Thiemann. On the decidability of subtyping with bounded existential types (extended edition). Technical report, Universität Freiburg, Sept. 2009. `ftp://ftp.informatik.uni-freiburg.de/documents/reports/report250/report00250.ps.gz`.