# Java type unification with wildcards

Martin Plümicke

University of Cooperative Education Stuttgart/Horb
Florianstraße 15, D–72160 Horb
m.pluemicke@ba-horb.de

**Abstract.** With the introduction of Java 5.0 the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. The term

`Vector<? extends Vector<AbstractList<Integer>>>`

is for example a correct type in Java 5.0.
In this paper we present a type unification algorithm for Java 5.0 type terms. The algorithm unifies type terms, which are in subtype relationship. For this we define Java 5.0 type terms and its subtyping relation, formally.
As Java 5.0 allows wildcards as instances of generic types, the subtyping ordering contains infinite chains. We show that the type unification is still finitary. We give a type unification algorithm, which calculates the finite set of general unifiers.

## 1 Introduction

With the introduction of Java 5.0 [1] the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. For example the term

`Vector<? extends Vector<AbstractList<Integer>>>`

is a correct type in Java 5.0.
Considering all that, it is often rather difficult for a programmer to recognize whether such a complex type is the correct one for a given method or not.
This has caused us to develop a Java 5.0 type inference system which assists the programmer by calculating types automatically. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principal types.
Following the ideas of [2], we reduce the Java 5.0 *type inference problem* to a Java 5.0 *type unification problem*. The Java 5.0 *type unification problem* is given as: For two type terms $\theta_1, \theta_2$ a substitution is demanded, such that

$\sigma(\theta_1) \leq^* \sigma(\theta_2)$, where $\leq^*$ is the Java 5.0 subtyping relation.

The type system of Java 5.0 is very similar to the type system of polymorphically order-sorted types, which is considered for the logical languages [3–6] and for the functional object–oriented language OBJ–P [7]. But in all approaches the type unification problem was not solved completely.

In [8] we have done a first step solving the type unification problem. We restricted the set of type terms, by disallowing wildcards, and presented a type unification algorithm for this approach. This algorithm led to the Java 5.0 type inference system presented in [9].
In this paper we extend our algorithm to type terms with wildcards. This means that we solve the original type unification problem and give a complete type unification algorithm. We will show, that the type unification problem is not still unitary, but finitary.
The paper is organized as follows. In the second section we formally describe the Java 5.0 type system including its inheritance hierarchy. In the third section we give an overview of the type unification problem. Then, we present the type unification algorithm and give an example. Finally, we close with a summary and an outlook.

## 2 Subtyping in Java 5.0

The Java 5.0 types are given as type terms over a type signature $TS$ of class/interface names and a set of bounded type variables $BTV$. While the type signature describes the arities of the the class/interface names, a bound of a type variable restricts the allowed instantiated types to subtypes of the bound. For a type variable `a` bounded by the type $ty$ we will write $\mathtt{a}|_{ty}$.

*Example 1.* Let the following Java 5.0 program be given:

```
class A<a> implements I<a> { ... }
class B<a> extends A<a> { ... }
class C<a extends I<b>,b> { ... }
interface I<a> { ... }
interface J<a> { ... }
class D<a extends B<a> & J<b>, b> { ... }
```

Then, the corresponding type signature $TS$ is given as: $TS^{(\mathtt{a}|_{\mathtt{Object}})} = \{\, \mathtt{A}, \mathtt{B}, \mathtt{I}, \mathtt{J} \,\}$, $TS^{(\mathtt{a}|_{\mathtt{I<b>}}\ \mathtt{b}|_{\mathtt{Object}})} = \{\, \mathtt{C} \,\}$, and $TS^{(\mathtt{a}|_{\mathtt{B<a>\&J<b>}}\ \mathtt{b}|_{\mathtt{Object}})} = \{\, \mathtt{D} \,\}$.
As $\mathtt{A}, \mathtt{I} \in TS^{(\mathtt{a}|_{\mathtt{Object}})}$ and `Integer` is a subtype of `Object`, the terms `A<Integer>` and `I<Integer>` are Java 5.0 types. As `I<Integer>` is a subtype of itself and `A<Integer>` is also a subtype of `I<Integer>`, the terms `C<I<Integer>, Integer>` and `C<A<Integer>, Integer>` are also type terms. But as `J<Integer>` is no subtype of `I<Integer>` the term `C<J<Integer>, Integer>` is no Java 5.0 type.

For the definition of the inheritance hierarchy, the concept of Java 5.0 *simple types* and the concept of *capture conversion* is needed. For the definition of Java 5.0 simple types we refer to [1], Section 4.5, where Java 5.0 parameterized types are defined. We call them, in this paper, Java 5.0 *simple types* in contrast to the function types of methods. Java 5.0 simple types consists of the Java 5.0 types as in Example 1 presented and wildcard types like `Vector<? extends Integer>`. For the definition of the *capture conversion* we refer to [1] §5.1.10. The *capture*

*conversion* transforms types with wildcard type arguments to equivalent types, where the wildcards are replaced by implicit type variables. The capture conversion of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ is denoted by $CC(\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}\,)$.

The inheritance hierarchy consists of two different relations: The "extends relation" (denoted by $<$) is explicitely defined in Java 5.0 programs by the *extends*, and the *implements* declarations, respectively. The "subtyping relation" (cp. [1], Section 4.10) is built as the reflexive, transitive, and instantiating closure of the extends relation.
In the following we will use ${}_?\theta$ as an abbreviation for the type term "`? extends` $\theta$" and ${}^?\theta$ as an abbreviation for the type term "`? super` $\theta$".

**Definition 1 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(\, BTV\,)$).** *Let $TS$ be a type signature of a given Java 5.0 program and $<$ the corresponding extends relation. The* subtyping relation $\leq^*$ *is given as the reflexive and transitive closure of the smallest relation satisfying the following conditions:*

- *if $\theta < \theta'$ then $\theta \leq^* \theta'$.*
- *if $\theta_1 \leq^* \theta_2$ then $\sigma_1(\,\theta_1\,) \leq^* \sigma_2(\,\theta_2\,)$ for all substitutions $\sigma_1$, $\sigma_2 : BTV \to \mathsf{SType}_{TS}(\, BTV\,)$, where for each type variable $a$ of $\theta_2$ holds $\sigma_1(\,a\,) = \sigma_2(\,a\,)$ (soundness condition).*
- *$a \leq^* \theta_i$ for $a \in BTV^{(\theta_1 \& \ldots \& \theta_n)}$ and $1 \leqslant i \leqslant n$*
- *It holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ if for each $\theta_i$ and $\theta'_i$, respectively, one of the following conditions is valid:*
  - *$\theta_i = {}_?\overline{\theta}_i$, $\theta'_i = {}_?\overline{\theta}'_i$ and $\overline{\theta}_i \leq^* \overline{\theta}'_i$.*
  - *$\theta_i = {}^?\overline{\theta}_i$, $\theta'_i = {}^?\overline{\theta}'_i$ and $\overline{\theta}'_i \leq^* \overline{\theta}_i$.*
  - *$\theta_i, \theta'_i \in \mathsf{SType}_{TS}(\, BTV\,)$ and $\theta_i = \theta'_i$*
  - *$\theta'_i = {}_?\theta_i$*
  - *$\theta'_i = {}^?\theta_i$ (cp. [1] §4.5.1.1 type argument containment)*
- *Let $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>}$ be the capture conversions of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ and $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ then holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.*

It is surprising that the condition for $\sigma_1$ and $\sigma_2$ in the second item is not $\sigma_1(\,a\,) \leq^* \sigma_2(\,a\,)$, but $\sigma_1(\,a\,) = \sigma_2(\,a\,)$. This is necessary in order to get a sound type system. This property is the reason for the introduction of wildcards in Java 5.0 (cp. [1], §5.1.10).
The next example illustrates the subtyping definition.

*Example 2.* Let the Java 5.0 program from Example 1 be given again. Then the following relationships hold:

- `A<a>` $\leq^*$ `I<a>`, as `A<a>` $<$ `I<a>`
- `A<Integer>` $\leq^*$ `I<Integer>`, where $\sigma_1 = [a \mapsto$ `Integer`$] = \sigma_2$
- `A<Integer>` $\leq^*$ `I<? extends Object>`, as `Integer` $\leq^*$ `Object`
- `A<Object>` $\leq^*$ `I<? super Integer>`, as `Integer` $\leq^*$ `Object`

There are elements of the extends relation, where the sub-terms of a type term are not variables. As elements like this must be handled especially during the unification (*adapt* rules, Fig. 1), we declare a further ordering on the set of type terms which we call the *finite closure* of the extends relation.

**Definition 2 (Finite closure of $<$).** *The* finite closure $\mathbf{FC}(<)$ *is the reflexive and transitive closure of pairs in the subtyping relation* $\leq^*$ *with* $C(a_1 \ldots a_n) \leq^* D(\theta_1, \ldots, \theta_m)$*, where the* $a_i$ *are type variables and the* $\theta_i$ *are type terms.*
*If a set of bounded type variables BTV is given, the finite closure* $\mathbf{FC}(<)$ *is extended to* $\mathbf{FC}(<, BTV)$*, by* $a|_\theta \leq^* a|_\theta$ *for* $a|_\theta \in BTV$*.*

**Lemma 1.** *The* finite closure $\mathbf{FC}(<)$ *is finite.*

Now we give a further example to illustrate the definition of the subtyping relation and the finite closure.

*Example 3.* Let the following Java 5.0 program be given.

```
abstract class AbstractList<a> implements List<a> {...}
class Vector<a> extends AbstractList<a> {...}
class Matrix<a> extends Vector<Vector<a>> {...}
```

Following the soundness condition of the Java 5.0 type system we get

$$\texttt{Vector<Vector<a>>} \not\leq^* \texttt{Vector<List<a>>},$$

but

$$\texttt{Vector<Vector<a>>} \leq^* \texttt{Vector<? extends List<a>>}.$$

The finite closure $\mathbf{FC}(<)$ is given as the reflexive and transitive closure of

$$\{\ \texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}$$
$$\texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>} \leq^* \texttt{AbstractList<Vector<a>>}$$
$$\leq^* \texttt{List<Vector<a>>}\ \}.$$

## 3 Type Unification

In this section we consider the type unification problem of Java 5.0 type terms. The type unification problem is given as: For two type terms $\theta_1, \theta_2$ a substitution is demanded, such that

$$\sigma(\theta_1) \leq^* \sigma(\theta_2).$$

The algorithm solving the type unification problem is an important basis of the Java 5.0 type inference algorithm.

### 3.1 Overview

First we give an overview of approaches considering the type unification problem. on polymorphically order-sorted types. In [3] the type unification problem is mentioned as an open problem. For the logical language TEL in [3] an incomplete type inference algorithm is given. The incompleteness is caused by the fact, that subtype relationships of polymorphic types which have different arities (e.g. `List(a)` $<$ `myLi(a,b)`) are allowed. This leads to the property that there are infinite chains in the type term ordering. Nevertheless, the type unification fails

in some cases without infinite chains, although there is a unifier. Let for example $\texttt{nat} < \texttt{int}$, and the set of unequations $\{\, \texttt{nat} \lessdot \texttt{a}, \texttt{int} \lessdot a \,\}$ be given, then $\{\, \texttt{a} \mapsto \texttt{nat} \,\}$ is determined, such that $\{\, \texttt{int} \lessdot \texttt{nat} \,\}$ fails, although $\{\, \texttt{a} \mapsto \texttt{int} \,\}$ is a unifier. For $\{\, \texttt{int} \lessdot \texttt{a}, \texttt{nat} \lessdot a \,\}$ the algorithm determines the correct unifier $\{\, \texttt{a} \mapsto \texttt{int} \,\}$.

In the typed logic programs of [5] subtype relationships of polymorphic types are allowed only between type constructors of the same arity. In this approach a *most general type unifier (mgtu)* is defined as an upper bound of different principal type unifiers. In general there are no upper bounds of two given type terms in the type term ordering, which means that there are in general no mgtu in the sense of [5]. For example for $\texttt{nat} < \texttt{int}$, $\texttt{neg} < \texttt{int}$, and the set of unequations $\{\, \texttt{nat} \lessdot \texttt{a},\ \texttt{neg} \lessdot \texttt{a} \,\}$ the mgtu $\{\, \texttt{a} \mapsto \texttt{int} \,\}$ is determined. If the type term ordering is extended by $\texttt{int} < \texttt{index}$ and $\texttt{int} < \texttt{expr}$, then there are three unifiers $\{\, \texttt{a} \mapsto \texttt{int} \,\}$, $\{\, \texttt{a} \mapsto \texttt{index} \,\}$, and $\{\, \texttt{a} \mapsto \texttt{expr} \,\}$, but none of them is a mgtu in the sense of [5].

The type system of PROTOS-L [6] was derived from TEL by disallowing any explicite subtype relationships between polymorphic type constructors.

In [6] a complete type unification algorithm is given, which can be extended to the type system of [5]. They solved the type unification problem for type term orderings following the restrictions of PROTOS-L respectively the restrictions of [5]. Additionally, the result of this paper is, that the type unification problem is not unitary, but finitary. This means in general that there is more than one general type unifier. For the above example the algorithm determines where $\texttt{nat} < \texttt{int}$, $\texttt{neg} < \texttt{int}$, $\texttt{int} < \texttt{index}$, and $\texttt{int} < \texttt{expr}$ and the set of unequations $\{\, \texttt{nat} \lessdot \texttt{a}, \texttt{neg} \lessdot \texttt{a} \,\}$ is given, the three general unifiers $\{\, \texttt{a} \mapsto \texttt{int} \,\}$, $\{\, \texttt{a} \mapsto \texttt{index} \,\}$, and $\{\, \texttt{a} \mapsto \texttt{expr} \,\}$.

Finally, in [8] we disallowed wildcards, which means that there is no subtyping in the arguments of the type term (soundness condition, Def. 1), but subtype relationship of type constructors with different arities is allowed. For type term orderings following this restriction we presented a type unification algorithm and proved that the type unification problem is also finitary. For example for $\texttt{myLi<b, a>} < \texttt{List<a>}$ and the set of unequations $\{\, \texttt{myLi<Integer, a>} \lessdot \texttt{List<Boolean>} \,\}$ the general unifier $\{\, \texttt{a} \mapsto \texttt{Boolean} \,\}$ is determined. For $\{\, \texttt{myLi<Integer, Integer>} \lessdot \texttt{List<Number>} \,\}$ the algorithm fails, as `Integer` is indeed a subtype of `Number`, but subtyping in the arguments is prohibited.

The type systems of TEL, respectively the other logical languages, and of Java 5.0 are very similar. The Java 5.0 type system has the same properties considering type unification, if it is restricted to simple types without parameter bounds (but including the wildcard constructions). The only difference is, that in TEL the number of arguments of a supertype type can be greater, whereas in Java 5.0 the number of arguments of a subtype can be greater. This means that infinite chains have a lower bound in TEL and an upper bound in Java 5.0. Let us consider the following example: In TEL for $\texttt{List(a)} \leq \texttt{myLi(a,b)}$ holds:

$$\texttt{List(a)} \leq \texttt{myLi(a,List(a))} \leq \texttt{myLi(a,myLi(a,List(a)))} \leq \ldots$$

In contrast in Java 5.0 for $\texttt{myLi<b, a>} < \texttt{List<a>}$ holds:

$$\ldots \leq^* \texttt{myLi<}_?\texttt{myLi<}_?\texttt{List<a>,a>,a>} \leq^* \texttt{myLi<}_?\texttt{List<a>,a>} \leq^* \texttt{List<a>}$$

The open type unification problem of [3] is caused by these infinite chains. We will now present a solution for the open problem.

Our type unification algorithm bases on the algorithm by A. Martelli and U. Montanari [10] solving the original untyped unification problem. The main difference is, that in the original unification an unifier $\sigma$ is demanded, such that $\sigma(\theta_1) = \sigma(\theta_2)$, whereas in our case an unifier $\sigma$ is demanded, such that $\sigma(\theta_1) \leq^* \sigma(\theta_2)$. This means, that in the original case a result of the algorithm $a = \theta$ leads to one unifier $[a \mapsto \theta]$. In contrast to that, a result $a \leq^* \theta$ leads to a set of unifiers $\{ [a \mapsto \overline{\theta}] \mid \overline{\theta} \leq^* \theta \}$ in our algorithm.

### 3.2 Type unification algorithm

In the following we denote $\theta \lessdot \theta'$ for two type terms, which should be type unified. During the unification algorithm $\lessdot$ is replaced by $\lessdot_?$ and $\doteq$, respectively. $\theta \lessdot_? \theta'$ means that the two sub-terms $\theta$ and $\theta'$ of type terms should be unified, such that $\sigma(\theta)$ is a subtype of $\sigma(\theta')$. $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.

In the following, the type unification algorithm is shown.

(adapt) $$\frac{Eq \cup \{ D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \}}{Eq \cup \{ D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>}[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n] \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \}}$$

where there are $\overline{\theta}'_1, \ldots, \overline{\theta}'_m$ with

- $(D\texttt{<}a_1, \ldots, a_n\texttt{>} \leq^* D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>}) \in \mathbf{FC}(<)$

(adaptExt) $$\frac{Eq \cup \{ D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot_? {}_?D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \}}{Eq \cup \{ D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>}[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n] \lessdot_? {}_?D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \}}$$

where $D \in \Theta^{(n)}$ or $D = {}_?\overline{X}$ with $\overline{X} \in \Theta^{(n)}$ and there are $\overline{\theta}'_1, \ldots, \overline{\theta}'_m$ with

- ${}_?D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>}) \in \mathbf{grArg}(D\texttt{<}a_1, \ldots, a_n\texttt{>})$

(adaptSup) $$\frac{Eq \cup \{ D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \lessdot_? {}^?D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \}}{Eq \cup \{ D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>}[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n] \lessdot_? {}_?D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \}}$$

where $D' \in \Theta^{(n)}$ or $D' = {}^?\overline{X}$ with $\overline{X} \in \Theta^{(n)}$ and there are $\overline{\theta}'_1, \ldots, \overline{\theta}'_m$ with

- ${}^?D\texttt{<}a_1, \ldots, a_n\texttt{>} \in \mathbf{grArg}(D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>})$

**Fig. 1.** Java 5.0 type unification adapt rules

(reduceUp) $\dfrac{Eq \cup \{\, \theta \ll {}^{?}\theta' \,\}}{Eq \cup \{\, \theta \ll \theta' \,\}}$ (reduceUpLow) $\dfrac{Eq \cup \{\, {}_{?}\theta \ll {}^{?}\theta' \,\}}{Eq \cup \{\, \theta \ll \theta' \,\}}$

(reduceLow) $\dfrac{Eq \cup \{\, {}_{?}\theta \ll \theta' \,\}}{Eq \cup \{\, \theta \ll \theta' \,\}}$

(reduce1) $\dfrac{Eq \cup \{\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \ll D\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \,\}}{Eq \cup \{\, \theta_{\pi(1)} \ll_? \theta'_1, \ldots, \theta_{\pi(n)} \ll_? \theta'_n \,\}}$

where
- $C\texttt{<}a_1, \ldots, a_n\texttt{>} \leq^* D\texttt{<}a_{\pi(1)}, \ldots, a_{\pi(n)}\texttt{>}$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- $\pi$ is a permutation

(reduceExt) $\dfrac{Eq \cup \{\, X\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \ll_? {}_{?}Y\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \,\}}{Eq \cup \{\, \theta_{\pi(1)} \ll_? \theta'_1, \ldots, \theta_{\pi(n)} \ll_? \theta'_n \,\}}$

where
- ${}_{?}Y\texttt{<}a_{\pi(1)}, \ldots, a_{\pi(n)}\texttt{>} \in \mathbf{grArg}(\, X\texttt{<}a_1, \ldots, a_n\texttt{>}\,)$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- $\pi$ is a permutation
- $X \in \Theta^{(n)}$ or $X = {}_{?}\overline{X}$ with $\overline{X} \in \Theta^{(n)}$.

(reduceSup) $\dfrac{Eq \cup \{\, X\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \ll_? {}^{?}Y\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \,\}}{Eq \cup \{\, \theta'_1 \ll_? \theta_{\pi(1)}, \ldots, \theta'_n \ll_? \theta_{\pi(n)} \,\}}$

where
- ${}^{?}Y\texttt{<}a_{\pi(1)}, \ldots, a_{\pi(n)}\texttt{>} \in \mathbf{grArg}(\, X\texttt{<}a_1, \ldots, a_n\texttt{>}\,)$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- $\pi$ is a permutation
- $X \in \Theta^{(n)}$ or $X = {}^{?}\overline{X}$ with $\overline{X} \in \Theta^{(n)}$.

(reduceEq) $\dfrac{Eq \cup \{\, X\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \ll_? X\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \,\}}{Eq \cup \{\, \theta_{\pi(1)} \doteq \theta'_1, \ldots, \theta_{\pi(n)} \doteq \theta'_n \,\}}$

(reduce2) $\dfrac{Eq \cup \{\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \doteq C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \,\}}{Eq \cup \{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}}$

where
- $C \in \Theta^{(n)}$ or $C = {}_{?}\overline{C}$ or $C = {}^{?}\overline{C}$ with $\overline{C} \in \Theta^{(n)}$, respectively.

(erase1) $\dfrac{Eq \cup \{\, \theta \ll \theta' \,\}}{Eq}\ \theta \leq^* \theta'$ (erase2) $\dfrac{Eq \cup \{\, \theta \ll_? \theta' \,\}}{Eq}\ \theta' \in \mathbf{grArg}(\, \theta \,)$

(erase3) $\dfrac{Eq \cup \{\, \theta \doteq \theta' \,\}}{Eq}\ \theta = \theta'$ (swap) $\dfrac{Eq \cup \{\, \theta \doteq a \,\}}{Eq \cup \{\, a \doteq \theta \,\}}\ \theta \notin BTV,\ a \in BTV$

**Fig. 2.** Java 5.0 type unification rules with wildcards

The type unification algorithm is given as follows

**Input:** Set of equations $Eq = \{\, \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \,\}$
**Precondition:** $\theta_i, \theta'_i \in \mathsf{SType}_{TS}(\, BTV \,)$ for $1 \leqslant i \leqslant n$.
**Output:** Set of all general type unifiers $Uni = \{\, \sigma_1, \ldots, \sigma_m \,\}$
**Postcondition:** For all $1 \leqslant i \leqslant n$ and for all $1 \leqslant j \leqslant m$ holds $(\sigma_j(\, \theta_i \,) \leq^* \sigma_j(\, \theta'_i \,))$.

The algorithm itself is given in seven steps:

1. Repeated application of the *adapt* rules (fig. 1), the *reduce* rules, the *erase* rules and the *swap* rule (fig. 2) to all elements of $Eq$. The end configuration of $Eq$ is reached if for each element no rule is applicable.
2. $Eq'_1 =$ *Subset of pairs, where both type terms are type variables*
3. $Eq'_2 = Eq \backslash Eq'_1$
4. $Eq'_{set}$
$$
\begin{aligned}
= \{\, Eq'_1 \,\} \times \Big( & \bigotimes_{(a \lessdot \theta') \in Eq'_2} \{\, ([a \doteq \theta] \cup \sigma) \mid (\overline{\theta} \leq^* C\!<\!\theta_1, \ldots, \theta_n\!>) \in \mathbf{FC}(\, < \,), \\
& \qquad \overline{\theta}' \in \{\, C\!<\!\theta'_1, \ldots, \theta'_n\!> \\
& \qquad\qquad \mid \theta'_i \in \mathbf{grArg}(\, \theta_i \,), 1 \leqslant i \leqslant n \,\} \\
& \qquad \sigma \in \mathsf{Unify}(\, \overline{\theta}', \theta' \,), \\
& \qquad \theta \in \mathbf{smaller}(\, \sigma(\, \overline{\theta} \,) \,) \,\} \Big) \\
\times \Big( & \bigotimes_{(a \lessdot_? {}_?\theta') \in Eq'_2} \{\, ([a \doteq \theta] \cup \sigma) \mid (\overline{\theta} \leq^* C\!<\!\theta_1, \ldots, \theta_n\!>) \in \mathbf{FC}(\, < \,), \\
& \qquad \overline{\theta}' \in \{\, C\!<\!\theta'_1, \ldots, \theta'_n\!> \\
& \qquad\qquad \mid \theta'_i \in \mathbf{grArg}(\, \theta_i \,), 1 \leqslant i \leqslant n \,\} \\
& \qquad \sigma \in \mathsf{Unify}(\, \overline{\theta}', \theta' \,), \\
& \qquad \theta \in \mathbf{smArg}(\, \sigma(\, {}_?\overline{\theta} \,) \,) \,\} \Big) \\
\times \Big( & \bigotimes_{(a \lessdot_? {}^?\theta') \in Eq'_2} \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{smArg}(\, {}^?\theta' \,) \,\} \Big) \\
\times \Big( & \bigotimes_{(a \lessdot_? \theta') \in Eq'_2} \{\, [a \doteq \theta'] \,\} \Big) \\
\times \Big( & \bigotimes_{(\theta \lessdot a) \in Eq'_2} \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{greater}(\, \theta \,) \,\} \Big) \\
\times \Big( & \bigotimes_{({}_?\theta \lessdot_? a) \in Eq'_2} \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{grArg}(\, {}_?\theta \,) \,\} \Big) \\
\times \Big( & \bigotimes_{({}^?\theta \lessdot_? a) \in Eq'_2} \{\, ([a \doteq {}^?\theta'] \cup \sigma) \mid (\overline{\theta} \leq^* C\!<\!\theta_1, \ldots, \theta_n\!>) \in \mathbf{FC}(\, < \,), \\
& \qquad \overline{\theta}' \in \{\, C\!<\!\theta'_1, \ldots, \theta'_n\!> \\
& \qquad\qquad \mid \theta'_i \in \mathbf{grArg}(\, \theta_i \,), 1 \leqslant i \leqslant n \,\} \\
& \qquad \sigma \in \mathsf{Unify}(\, \overline{\theta}', \theta \,), \\
& \qquad \theta' \in \mathbf{smaller}(\, \sigma(\, \overline{\theta} \,) \,) \,\} \Big) \\
\times \Big( & \bigotimes_{(\theta \lessdot_? a) \in Eq'_2} \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{grArg}(\, \theta \,) \,\} \Big) \\
\times \{\, & [a \doteq \theta \mid (a \doteq \theta) \in Eq'_2] \,\}
\end{aligned}
$$
5. Application of the following *subst* rule

$$\text{(subst)}\ \frac{Eq' \cup \{\, a \doteq \theta \,\}}{Eq'[a \mapsto \theta] \cup \{\, a \doteq \theta \,\}}\ a \text{ occurs in } Eq' \text{ but not in } \theta$$

for each $a \doteq \theta$ in each element of $Eq' \in Eq'_{set}$.

6. (a) Foreach $Eq' \in Eq'_{set}$ which has changed in the last step start again with the first step.
   (b) Build the union $Eq''_{set}$ of all results of (a) and all $Eq' \in Eq'_{set}$ which has not changed in the last step.
7. $Uni = \{\, \sigma \mid Eq'' \in Eq''_{set}, Eq''$ is in solved form, $\sigma = \{\, a \mapsto \theta \mid (a \doteq \theta) \in Eq'' \,\} \,\}$

In the algorithm the unbounded wildcard "`?`" is denoted as the equivalent bounded wildcard "`? extends Object`".
Furthermore, there are functions **greater** and **grArg**, respectively **smaller** and **smArg**. All functions determine supertypes, respectively subtypes by pattern matching with the elements of the finite closure. The functions **greater** and **smaller** determine the supertypes respectively subtypes of simple types, while **grArg** and **smArg** determine the supertypes respectively the subtypes of sub-terms, which are allowed as arguments in type terms.
The function Unify is the ordinary unification.

Now we will explain the rules in Fig. 1 and 2.

**adapt rules:** The *adapt* rules adapt type term pairs, which are built by class declarations like

`class` $C$`<`$a_1, \ldots, a_n$`>` `extends` $D$`<`$D_1$`<`$\ldots$`>`$, \ldots, D_m$`<`$\ldots$`>>`.

The smaller type is replaced by a type term, which has the same outermost type name as the greater type. Its sub-terms are determined by the finite closure. The instantiations are maintained.
The *adaptExt* and *adaptSup* rule are the corresponding rules to the *adapt* rule for sub-terms of type terms.

**reduce rules:** The rules *reduceUp*, *reduceUpLow*, and *reduceLow* correspond to the extension of the subtyping ordering on extended simple types [11] (wildcard sub-terms of type terms).
The *reduce1* rule follows from the construction of the subtyping relation $\leq^*$, where from $C(\, a_1, \ldots, a_n \,) \;<\; D(\, a_{\pi(1)}, \ldots, a_{\pi(n)} \,)$ follows $C(\, \theta_1, \ldots, \theta_n \,) \leq^* D(\, \theta'_1, \ldots, \theta'_n \,)$ if and only if $\theta'_i \in \mathbf{grArg}(\, \theta_{\pi(i)} \,)$ for $1 \leqslant i \leqslant n$.
The *reduceExt* and the *reduceSup* rules are the corresponding rules to the *reduce1* rule for sub-terms of type terms.
The *reduceEq* and the *reduce2* rule ensures, that sub-terms must be equal, if there are no wildcards (soundness condition of the Java 5.0 type system).

**erase rules:** The *erase* rules erase type term pairs, which are in the respective relationship.

**swap rule:** The *swap* rule swaps type term pairs, such that type variables are mapped to type terms, not vice versa.

Now we give an example for the type unification algorithm.

*Example 4.* In this example we use the standard Java 5.0 types `Number`, `Integer`, `Stack`, `Vector`, `AbstractList`, and `List`. It holds `Integer` $<$ `Number` and `Stack<a>` $<$ `Vector<a>` $<$ `AbstractList<a>` $<$ `List<a>`.

As a start configuration we use

$$\{\,(\texttt{Stack<a>} \lessdot \texttt{Vector<}{}_{?}\texttt{Number>}),\ (\texttt{AbstractList<Integer>} \lessdot \texttt{List<a>})\,\}.$$

In the first step the reduce1 rule is applied twice:

$$\{\,\texttt{a} \lessdot_{?} {}_{?}\texttt{Number},\ \texttt{Integer} \lessdot_{?} \texttt{a}\,\}$$

With the second and the third step we receive in step four:

$$\begin{aligned}\{\ &\{\,\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}\{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\}\,\}\end{aligned}$$

In the fifth step the rule *subst* is applied:

$$\begin{aligned}\{\ &\{\,\texttt{Integer} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \underline{\{\,{}_{?}\texttt{Number} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\}},\\ &\{\,{}_{?}\texttt{Integer} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \underline{\{\,{}^{?}\texttt{Integer} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\}},\\ &\{\,\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,{}_{?}\texttt{Number} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,{}_{?}\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,{}^{?}\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{Integer} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,{}_{?}\texttt{Number} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\underline{\{\,{}_{?}\texttt{Integer} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}}, \{\,{}^{?}\texttt{Integer} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\underline{\{\,\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}}, \{\,{}_{?}\texttt{Number} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\underline{\{\,{}_{?}\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}}\{\,{}^{?}\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\}\,\}\end{aligned}$$

The underlined sets of type term pairs lead to unifiers.
Now we have to continue with the first step (step 6(a)). With the application of the *erase3* rule and step 7, we get:

$$\{\,\{\,a \mapsto {}_{?}\texttt{Number}\,\}, \{\,a \mapsto {}_{?}\texttt{Integer}\,\}, \{\,a \mapsto \texttt{Integer}\,\}\,\}.$$

The following example shows, how the algorithm works for subtype relationships with different numbers of arguments and wildcards, which causes infinite chains (cp. Section 3.1).

*Example 5.* Let $\texttt{myLi<b, a>} < \texttt{List<a>}$ and the start configuration $\{\,\texttt{List<x>} \lessdot \texttt{List<}{}_{?}\texttt{List<Integer>>}\,\}$ be given. In the first step the reduce 1 rule is applied: $\{\,\texttt{x} \lessdot_{?} {}_{?}\texttt{List<Integer>}\,\}$. With the second step we get the result:

$$\begin{aligned}\{\ &\{\,\texttt{x} \mapsto \texttt{List<Integer>}\,\}, \{\,\texttt{x} \mapsto {}_{?}\texttt{List<Integer>}\,\},\\ &\{\,\texttt{x} \mapsto \texttt{myLi<b, Integer>}\,\}, \{\,\texttt{x} \mapsto {}_{?}\texttt{myLi<b, Integer>}\,\}\,\}.\end{aligned}$$

All other inifite numbers of unifiers are instances of these general unifiers.

The following theorem shows, that the type unification problem is solved by the type unification algorithm.

**Theorem 1.** *The type unification algorithm determines exactly all general type unifiers for a given set of type term pairs. This means that the algorithm is sound and complete.*

Now we give a sketch of the proof.

*Proof.* We do the proof in two steps. First we prove the *soundness* and then the *completeness*.

**Soundness** For the proof of the soundness, we take a substitution which is a result of the algorithm. Then we show that this substitution is a general unifier of the algorithm input. We do this by proving for each transformation, that if the substitution is a general unifier of the result of the transformation then the substitution is also a general unifier of the input of the transformation.

**Completeness** For the completeness we prove that if there is a general unifier of a set of type term pairs then this general unifier is also determined by the algorithm. We prove this, by showing for each transformation of the algorithm, that if a substitution is the general unifier of a set of type terms before the transformation is done, then the substitution is also a general unifier of at least one set of type term pairs after the transformation.

**Step One:** For step one for each rule of Fig. 1 and of Fig. 2 is showed: If $\sigma$ is a general unifier before the application then $\sigma$ is still a general unifier after the application.

**Step two and three:** There is nothing to prove, as the type term pairs are only seperated in different sets.

**Step four:** There are different cases, which have to be considered. As the following hold, there are always at least one set of type term pairs after the application, where $\sigma$ is a general unifier after the application, if it has been a general unifier before the application: For $\sigma$ with $\sigma(\, a \,) \leq^* \sigma(\, \theta' \,)$, there is a $\theta \leq^* \theta'$, where $\sigma(\, a \,) = \sigma(\, \theta \,)$ respectively for $\sigma$ with $\sigma(\, \theta \,) \leq^* \sigma(\, a \,)$, there is a $\theta \leq^* \theta'$, where $\sigma(\, a \,) = \sigma(\, \theta' \,)$.

**Step five:** As for a general unifier $\sigma$ of $\{\, a \doteq \theta \,\}$ holds $\sigma(\, a[a \mapsto \theta] \,) = \sigma(\, \theta \,)$, the subst rule preserves the general unifiers.

**Step six and seven:** As the sets of type terms are unchanged the general unifiers are preserved.

As step four of the type unification algorithm is the only possibility, where the number of unifiers are multiplied, we can, according to lemma 1 and theorem 1, conclude as follows.

**Corollary 1 (Finitary).** *The type unification of Java 5.0 type terms with wildcards is finitary.*

**Corollary 2 (Termination).** *The type unification algorithm terminates.*

Corollary 1 means that the open problem of [3] is solved by our type unification algorithm.

## 4 Conclusion and Outlook

In this paper we presented an unification algorithm, which solves the type unification problem of Java 5.0 type terms with wildcards. Although the Java 5.0 subtyping ordering contains infinite chains, we showed that the type unification is finitary. This means that we solved the open problem from [3].
The Java 5.0 type unification is the base of the Java 5.0 type inference, as the usual unification is the base of type inference in functional programming languages.
We will extend our Java 5.0 type inference implementation [9] without wildcards by wildcards, which means that we have to substitute the Java 5.0 type unification algorithm without wildcards [8] by the new unification algorithm presented in this paper.

## References


1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java$^{TM}$ Language Specification. 3rd edn. The Java series. Addison-Wesley (2005)
2. Damas, L., Milner, R.: Principal type-schemes for functional programs. Proc. 9th Symposium on Principles of Programming Languages (1982)
3. Smolka, G.: Logic Programming over Polymorphically Order-Sorted Types. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany (May 1989)
4. Hanus, M.: Parametric order-sorted types in logic programming. Proc. TAPSOFT 1991 **LNCS**(394) (1991) 181–200
5. Hill, P.M., Topor, R.W.: A Semantics for Typed Logic Programs. In Pfenning, F., ed.: Types in Logic Programming. MIT Press (1992) 1–62
6. Beierle, C.: Type inferencing for polymorphic order-sorted logic programs. In: International Conference on Logic Programming. (1995) 765–779
7. Plümicke, M.: OBJ–P The Polymorphic Extension of OBJ–3. PhD thesis, University of Tuebingen, WSI-99-4 (1999)
8. Plümicke, M.: Type unification in Generic–Java. In Kohlhase, M., ed.: Proceedings of 18th International Workshop on Unification (UNIF'04). (July 2004)
9. Plümicke, M., Bäuerle, J.: Typeless Programming in Java 5.0. In Gitzel, R., Aleksey, M., Schader, M., Krintz, C., eds.: 4th International Conference on Principles and Practices of Programming in Java. ACM International Conference Proceeding Series, Mannheim University Press (August 2006) 175–181
10. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems **4** (1982) 258–282
11. Plümicke, M.: Formalization of the Java 5.0 type system. In: 24. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, Bad Honnef, Christian–Albrechts–Universität, Kiel (2. - 4. Mai 2007) (to appear).