Java type unification with wild cards

Martin Pluemicke

▶ **To cite this version:**

Martin Pluemicke. Java type unification with wild cards. UNIF07, 2007, Paris, France. inria-00175937

HAL Id: inria-00175937
https://inria.hal.science/inria-00175937

Submitted on 1 Oct 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Java type unification with wildcards

Martin Plümicke

University of Cooperative Education Stuttgart/Horb
Department of Information Technology
Florianstraße 15
D–72160 Horb
tel. +49-7451-521142
fax. +49-7451-521190
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 principle types.
Following the ideas of [2], we reduce the Java 5.0 type inference problem to a Java 5.0 type unification problem. In [2] a type inference algorithm for a core SML [3] bases on Robinson's unification algorithm [4]. The Java 5.0 *type unification*

problem is given as: For two type terms θ_1, θ_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 language TEL [5] and for the functional object–oriented language OBJ–P [6].
In [5] the type unification problem is mentioned as an open problem. This causes from the property, that in TEL subtype relationships between polymorphic types having different arities (e.g. `List(a)` $\leq$ `mytype(a,b)`) are allowed, which means that the subtyping relation contains infinite chains. In Java 5.0 subtype relationships as `mytype<a, b>` $\leq^*$ `Vector<a>` are also allowed. This means that there is a similar problem.
In [7] we solved the type unification problem for a restricted set of type terms. We considered only type terms without wildcards. This restriction guarantees that in the subtyping relation there are no infinite chains.
In this paper we extend our algorithm to type terms with wildcards. The means that the subtyping relation contains infinite chains. In this paper we show, that the type unification is still finitary. Our type unification algorithm calculates the finite set of general type unifiers.
This finitary property of the type unification leads to the property of the corresponding type inference algorithm, that sometimes more than one typing is inferred for Java 5.0 methods (cp. [8]).
The paper is organized as follows. In the second section we formally describe the Java 5.0 type system including its inheritance hierarchy. In section three we present the type unification algorithm and give some examples. Finally, we close with a summary and an outlook.

2 Java 5.0 Types

The base of the types are elements of the set of terms $T_\Theta(\,TV\,)$, which are given as a set of terms over a finite rank alphabet $\Theta = \Theta^{(n)}_{n\in N}$ of class names and a set of type variables TV. Therefore we denote them as *type terms* instead of types.

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> { ... }
```

The rank alphabet $\Theta = \Theta^{(n)}_{n \in N}$ is determined by

$$\Theta^{(1)} = \{\, \mathtt{A}, \mathtt{B}, \mathtt{I}, \mathtt{J} \,\} \text{ and } \Theta^{(2)} = \{\, \mathtt{C}, \mathtt{D} \,\}.$$

For example `A<Integer>`, `A<B<Boolean>>`, and `C<A<Object>, Object>` are type terms.

As the type terms are constructed over the class names, we call the class names in this framework *type constructors*.
If we consider the Java 5.0 program of Example 1 more accurately, we recognize that the bounds of the type parameters `b` in the class `C` and the bounds of the type parameter `a` in the class `D` are not considered. This leads to the problem that type terms like `C<C<a>, a>` are in the term set $T_\Theta(\, TV \,)$, although they are not correct in Java 5.0.
The solution of the problem is the extension of the rank alphabet Θ to a type signature, where the arity of the type constructors is indexed by bounded type variables. This leads to a restriction in the type term construction, such that the correct set of type terms is a subset of $T_\Theta(\, TV \,)$. Additionally the set of correct type terms is added by some wildcard constructions. We call the set of correct types *set of simple types* $\mathsf{SType}_{TS}(\, BTV \,)$ (Def. 3).
Unfortunately, the definitions of the type signature (Def. 2), the simple types (Def. 3), and the subtyping ordering (Def. 5) are mutually dependent. This is caused by the fact, that the restriction of the set of simple types is defined by bounded type parameters, whose bounds are also simple types. This means that, for some definitions, we must assume a given set of simple types, without knowing, how the set of simple types is exactly defined.

Definition 1 (Bounded type variables). *Let* $\mathsf{SType}_{TS}(\, BTV \,)$ *be a set of simple types. Then, the set of* bounded type variables *is an indexed set* $BTV = (BTV^{(ty)})_{ty \in \mathsf{I}(\, \mathsf{SType}_{TS}(\, BTV \,)\,)}$*, where each type variable is assigned to an intersection of simple types.*
$\mathsf{I}(\, \mathsf{SType}_{TS}(\, BTV \,)\,)$ *denotes the set of intersections over simple types (cp. Def. 3). In the following we will write a type variable* `a` *bounded by the type* ty *as* $\mathtt{a}|_{ty}$*. Type variables which are not bounded can be considered as bounded type variables by* `Object`*.*

Example 2. Let the following Java 5.0 class be given.

```
class BoundedTypeVars<a extends Number> {
    <t extends Vector<Integer> & J<a> & I,
     r extends Number> void m ( ... ) { ... }
}
```

The set of bounded type variable BTV of the method `m` is given as $BTV^{(\texttt{Number})} = \{\, \mathtt{a}, \mathtt{r} \,\}$ and $BTV^{(\texttt{Vector<Integer> \& J<a> \& I})} = \{\, \mathtt{t} \,\}$.

Definition 2 (Type signature, type constructor). *Let* $\mathsf{SType}_{TS}(\, BTV \,)$ *be a set of simple types. A* type signature TS *is a pair* $(\mathsf{SType}_{TS}(\, BTV \,),\ TC)$ *where* BTV *is an indexed set of bounded type variables and* TC *is a* $(BTV)^*$*–indexed set of* type constructors *(class names).*

Example 3. Let the Java 5.0 program from Example 1 be given again. Then, the corresponding indexed set of type constructors is given as $TC^{(\texttt{a}|_{\texttt{Object}})} = \{\,\texttt{A}, \texttt{B}, \texttt{I}, \texttt{J}\,\}$, $TC^{(\texttt{a}|_{\texttt{I<b>}}\ \texttt{b}|_{\texttt{Object}})} = \{\,\texttt{C}\,\}$, and $TC^{(\texttt{a}|_{\texttt{B<a>\&J<b>}}\ \texttt{b}|_{\texttt{Object}})} = \{\,\texttt{D}\,\}$.

In order to define the set of simple types, we have to introduce the *implicit type variables* with lower and upper bounds first. Implicite type variables are used in Java 5.0 during the *capture conversion* (Def. 4), where the wildcards are replaced by implicit type variables. Implicit type variables cannot be used explictly in Java 5.0 programs.
We denote an implicit type variable T with a lower bound ty by ${}_{ty}|T$ and with an upper bound ty' by $T|^{ty'}$.
The following definition of the set of simple types is connected to the corresponding definition in [1], section 4.5.

Definition 3 (Simple types). *The set of* simple types $\mathsf{SType}_{TS}(\,BTV\,)$ *for a given type signature* $(\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *is defined as the smallest set satisfying the following conditions:*

- *For each intersection type ty:* $\underline{BTV^{(ty)}} \subseteq \mathsf{SType}_{TS}(\,BTV\,)$
- $\underline{TC^{()}} \subseteq \mathsf{SType}_{TS}(\,BTV\,)$
- *For* $ty_i \in \mathsf{SType}_{TS}(\,BTV\,)$
  $\cup\,\{\,\texttt{?}\,\}$
  $\cup\,\{\,\texttt{? extends}\ \tau \mid \tau \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$
  $\cup\,\{\,\texttt{? super}\ \tau \mid \tau \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$
  and $C \in TC^{(a_1|_{b_1}\ldots a_n|_{b_n})}$ *holds*

$$\underline{C\texttt{<}ty_1, \ldots, ty_n\texttt{>}} \in \mathsf{SType}_{TS}(\,BTV\,)$$

  if after $C\texttt{<}ty_1, \ldots, ty_n\texttt{>}$ *subjected to the capture conversion (Def. 4) resulting in the type* $C\texttt{<}\overline{ty_1}, \ldots, \overline{ty_n}\texttt{>}$[1]*, for each actual type argument* $\overline{ty_i}$ *holds:*

$$\overline{ty_i} \leq^* b_i[a_j \mapsto \overline{ty_j} \mid 1 \leqslant j \leqslant n],$$

  where $\leq^*$ *is a subtyping ordering (Def. 5).*
- *The* $\underline{\text{set of implicit type variables with lower or upper bounds}}$ *belongs to* $\mathsf{SType}_{TS}(\,BTV\,)$

The set of intersection types *over a set of* $\mathsf{SType}_{TS}(\,BTV\,)$ *is denoted by:*
$\mathsf{I}(\,\mathsf{SType}_{TS}(\,BTV\,)\,) = \{\,\theta_1 \,\&\, \ldots \,\&\, \theta_n \mid \theta_i \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$

The following example shows the simple type construction, where the arguments of the type constructors are unbounded, respectively, bounded by `Object`.

Example 4. Let the Java 5.0 program from Example 1 and the corresponding indexed set of type constructors TC from Example 3 be given again. Let additional $\texttt{Integer} \in TC^{()}$.

[1] For non wildcard type arguments the capture conversion $\overline{ty_i}$ equals ty_i

The terms `A<Integer>` and `A<I<Integer>>` are simple types.
From $\texttt{Integer} \in TC^{()}$ follows `Integer` is a simple type. As $\texttt{A} \in TC^{(\texttt{a}|_{\texttt{Object}})}$ with $ty_1 = \texttt{Integer}$ follows, that `A<Integer>` is a simple type. From this follows as $\texttt{I} \in TC^{(\texttt{a}|_{\texttt{Object}})}$ with $ty_1 = \texttt{I<Integer>}$, that `A<I<Integer>>` is also a simple type.

After the definitions of the capture conversion and the subtyping relation, we give another example, where the arguments of the type constructors are bounded and wildcards are used.
In the following we will use ${}_{?}\theta$ as an abbreviation for the type term "`? extends` θ" and ${}^{?}\theta$ as an abbreviation for the type term "`? super` θ".

Before we can define the *subtyping relation* on simple types, we have to give the defintion of the *capture conversion* (cf. [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.

Definition 4 (Capture conversion). *Let* $TS = (\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *be a type signature. Furthermore, let be* $C \in TC^{(a_1|_{u_1},\ldots,a_n|_{u_n})}$ *and* $C\texttt{<}\theta_1,\ldots,\theta_n\texttt{>} \in \mathsf{SType}_{TS}(\,BTV\,)$*. Thus, the* capture conversion $C\texttt{<}\overline{\theta}_1,\ldots,\overline{\theta}_n\texttt{>}$ *of* $C\texttt{<}\theta_1,\ldots,\theta_n\texttt{>}$ *is defined as:*

- *if* $\theta_i = \,?$ *then* $\overline{\theta}_i = b_i|^{u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n]}$*, where* b_i *is a fresh implicit type variable.*
- *if* $\theta_i = {}_{?}\theta'_i$ *then* $\overline{\theta}_i = b_i|^{\theta'_i \,\&\, u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n]}$*, where* b_i *is a fresh implicit type variable with* upper bound $\theta'_i \,\&\, u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n])$*.*
- *if* $\theta_i = {}^{?}\theta'_i$ *then* $\overline{\theta}_i =_{\theta'_i} |b_i|^{u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n]}$*, where* b_i *is a fresh implicit type variable with* lower bound θ'_i *and* upper bound $u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n])$*.*
- *otherwise* $\overline{\theta}_i = \theta_i$

The capture conversion of $C\texttt{<}\theta_1,\ldots,\theta_n\texttt{>}$ *is denoted by* $CC(\,C\texttt{<}\theta_1,\ldots,\theta_n\texttt{>}\,)$*.*

Example 5. Let the indexed set of type constructors TC from Example 3 be given again. Then the following holds

$$CC(\,\texttt{A<? extends Integer>}\,) = \texttt{A<X}|^{\texttt{Integer}}\texttt{>}, \text{ as } \texttt{A} \in TC^{(\texttt{a}|_{\texttt{Object}})},$$
$$CC(\,\texttt{C<? extends A<a>, a>}\,) = \texttt{C<Y}|^{\texttt{A<a>\&I<a>}}, \texttt{a>}, \text{ as } C \in TC^{(\texttt{a}|_{\texttt{I<b>}}\ \texttt{b}|_{\texttt{Object}})},$$
$$CC(\,\texttt{B<? super Integer>}\,) = \texttt{B<}_{\texttt{Integer}}\texttt{|Z}|^{\texttt{Object}}\texttt{>}, \text{ as } B \in TC^{(\texttt{a}|_{\texttt{Object}})}.$$

The inheritance hierarchy consists of two different relations: The "extends relation" (in sign $<$) is explicitly 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.

Definition 5 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(\,BTV\,)$). *Let* $TS = (\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *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* σ_1, $\sigma_2 : BTV \to \mathsf{SType}_{TS}(BTV)$, *where for each type variable* a *of* θ_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* θ_i *and* θ'_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* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.
- *For an intersection type* $ty = \theta_1 \,\&\, \ldots \,\&\, \theta_n$ *holds* $ty \leq^* \theta_i$ *for any* $1 \leqslant i \leqslant n$.
- $T|^{(\theta_1 \& \ldots \& \theta_n)} \leq^* \theta_i$ *for any* $1 \leqslant i \leqslant n$.
- $\theta \leq^* {}_\theta|T$

It is surprising that the condition for σ_1 and σ_2 in the second item is not $\sigma_1(a) \leq^* \sigma_2(a)$, but $\sigma_1(a) = \sigma_2(a)$. This is necessary 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 6. 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 $\sigma_1 = [a \mapsto \texttt{Integer}] = \sigma_2$
- `A<Integer>` $\leq^*$ `I<? extends Object>`, as `Integer` $\leq^*$ `Object`
- `A<Object>` $\leq^*$ `I<? super Integer>`, as `Integer` $\leq^*$ `Object`

Finally, we give a further example for the construction of simple types, where the capture conversions and the subtyping relation are necessary.

Example 7. Let the type signature $(\mathsf{SType}_{TS}(BTV), TC)$ from Example 3 be given, which is derived from the Java 5.0 program of Example 1.

1. If we extend the Java 5.0 program by the class declaration

   `class BJ<a> extends B<BJ<a>> implements J<a> { ... }`

   then the type `D<BJ<Integer>, Integer>` is element of $\mathsf{SType}_{TS}(BTV)$. As `D` $\in TC^{(\texttt{a}|_{\texttt{B<a>\&J<b>}}\ \texttt{b}|_{\texttt{Object}})}$ the following must be valid:

   `BJ<Integer>` $\leq^*$ `B<a>`$[\texttt{a} \mapsto \texttt{BJ<Integer>}, \texttt{b} \mapsto \texttt{Integer}]$ $(=$ `B<BJ<Integer>>`$)$,

   `BJ<Integer>` $\leq^*$ `J<b>`$[\texttt{a} \mapsto \texttt{BJ<Integer>}, \texttt{b} \mapsto \texttt{Integer}]$ $(=$ `J<Integer>`$)$,

   and

   `Integer` $\leq^*$ `Object`$[\texttt{a} \mapsto \texttt{BJ<Integer>}, \texttt{b} \mapsto \texttt{Integer}]$ $(=$ `Object`$)$.

   This follows, as `BJ<a>` $\leq^*$ `B<BJ<a>>` and `BJ<a>` $\leq^*$ `J<a>`.

2. The type $\texttt{C<? extends A<a>, a>}$ is a simple type. As $\texttt{C} \in TC^{(\texttt{a}|_{\texttt{I<b>}}\ \texttt{b}|_{\texttt{Object}})}$ and $CC(\ \texttt{C<? extends A<a>, a>}\ ) = \texttt{C<X}|^{\texttt{A<a>\&I<a>}}, \texttt{a>}$ the following must be valid:
$$X|^{\texttt{A<a>\&I<a>}} \leq^* \texttt{I<b>}[\texttt{a} \mapsto X|^{\texttt{A<a>\&I<a>}}, \texttt{b} \mapsto \texttt{a}](= \texttt{I<a>})$$
and
$$\texttt{a} \leq^* \texttt{Object}[\texttt{a} \mapsto X|^{\texttt{A<a>\&I<a>}}, \texttt{b} \mapsto \texttt{a}](= \texttt{Object}).$$
This follows directly from the subtyping definition.
3. The given type term $\texttt{D<A<Integer>, Integer>}$ is no simple type. As $\texttt{D} \in TC^{(\texttt{a}|_{\texttt{B<a>\&J<b>}}\ \texttt{b}|_{\texttt{Object}})}$ it would be necessary that $\texttt{A<Integer>}$ is a subtype of $\texttt{B<A<Integer>>}$ as well as of $\texttt{J<Integer>}$.
From the declaration (cp. Example 1) it is obvious that $\texttt{A<Integer>}$ is neither a subtype of $\texttt{B<A<Integer>>}$ nor a subtype of $\texttt{J<Integer>}$.

As we consider elements of the *extends* relation, like $\texttt{BJ<c>} < \texttt{B<BJ<c>>}$, we recognize that 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. 3), we declare a further ordering on the set of type terms which we call the *finite closure* of the extends relation.

Definition 6 (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* θ_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 8. 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
$$\begin{aligned}\{\ &\texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}\\ &\texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>}\\ &\qquad\quad \leq^* \texttt{AbstractList<Vector<a>>}\\ &\qquad\quad \leq^* \texttt{List<Vector<a>>}\ \}.\end{aligned}$$

Remark 1. The type system of polymorphic order-sorted types, which is considered for a logical language TEL [5] and for a functional object–oriented language OBJ–P [9], is very similar to the Java 5.0 type system.

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 θ_1, θ_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.
In this section we will first present an overview of similar type unification problems. In the next part we will present the algorithm for the solution of the Java 5.0 type unification problem.

3.1 Overview

In the last section we assert, that the type system of Java 5.0 can be considered as polymorphic order-sorted types. Besides our functional language OBJ–P [9] for example in the logical languages TEL [5] and PROTOS-L [10] polymorphic order-sorted types are used. The logical language TEL allows subtype relationships between polymorphic types having different arities (e.g. `List(a)` $\leq$ `myLi(a,b)`). This implies that the subtyping relation contains infinite chains. The type system of PROTOS-L was derived from TEL by disallowing any explicite subtype relationship between polymorphic type constructors.
The given type inference algorithm and type unification algorithm, respectively, in [5] are incomplete. They are even incomplete for the restricted type system of [10].
In [5] the type unification problem of TEL without restrictions on the polymorphic type constructors is mentioned as an open problem.
If we compare the type system of TEL to the Java 5.0 type system, we assert that these type systems are very similar. The Java 5.0 type system restricted to simple types without parameter bounds (but including the wildcard constructions) has the same properties considering type unification. 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 in TEL inifinite chains has a lower bound and in Java 5.0 an upper bound. Let us consider the following example: In TEL for `List(a)` $\leq$ `myLi(a,b)` it 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 `myLi<b, a>` $<$ `List<a>` it holds:

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

The open type unification problem of [5] is caused by these infinite chains. We will present a solution for the open problem. The type unification problem is not longer unitary, but finitary, which means that there is more than one general unifier, but the number of general unifiers is finite.
Our type unification algorithm bases on the algorithm by A. Martelli and U. Montanari [11] solving the original untyped unification problem. There is another

unification algorithm in [4] which also solves the untyped unification problem. This algorithm is the basis of the ML type inference algorithm [12, 2].

3.2 Type unification algorithm

The basis of the type inference algorithm is the type unification. As said before, the *type unification problem* is given as: For two type terms θ_1, θ_2 a substitution is demanded, such that $\sigma(\theta_1) \leq^* \sigma(\theta_2)$. σ is called an unifier of θ_1 and θ_2. In the following we denote $\theta \lessdot \theta'$ for two type terms, which should be type unified. As said before, our type unification algorithm is based on the unification algorithm by Martelli and Montanari [11]. The main difference is, that in the original unification an unifier is demanded, such that $\sigma(\theta_1) = \sigma(\theta_2)$. This means that a pair $a \doteq \theta$ determines that the unifier substitutes a by the term θ. In contrast a pair $a \lessdot \theta$ and $\theta \lessdot a$, respectively, leads to multiple correct substitutions. All type terms smaller than θ and greater than θ, respectively, are correct substitutions for a. This is the reason, that there are multiple unifiers.
During the unification algorithm $\lessdot$ is replaced by $\lessdot_?$ and $\doteq$, respectively. $\theta \lessdot_? \theta'$ means that the two sub-terms θ and θ' of type terms should be unified, such that $\sigma(\theta)$ is a subtype of $\sigma(\theta')$, which are allowed as arguments in type terms, as defined in the fourth item of the subtyping definition (def. 5). $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.
Now, we give the type unification algorithm.

1. Repeated application of the *reduce* rules (fig. 1), the *erase* rules and the *swap* rule (fig. 2), and the *adapt* rules (fig. 3).
2. For each pair $a \lessdot \theta$ and $a \lessdot_? \theta$, respectively, for all subtypes $\overline{\theta}$ of θ, constructed by pattern–matching with elements from $\mathbf{FC}(\leq^*, BTV)$, pairs $a \doteq \overline{\theta}$ are built.
3. For each pair $\theta \lessdot a$ and $\theta \lessdot_? a$, respectively, for all supertypes θ' of θ, constructed by pattern–matching with elements from $\mathbf{FC}(\leq^*, BTV)$, pairs $a \doteq \theta'$ are built.
4. The cartesian product of the sets from step 2 and 3 is built.
5. Application of the following *subst* rule

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

6. For all changed sets of type terms start again with step 1.
7. Summerize all results.

Now, we will give an explaination for the rules in fig. 1, 2, and 3. There is a function **grArg**. The function determines the supertypes of sub-terms, which are allowed as arguments in type terms. Now, we consider the rules explicitly:

reduce rules: The rules *reduceUp*, *reduceUpLow*, and *reduceLow* erase leading wildcards, such that the reduction can be continued.

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

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

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

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

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

where
- ${}_{?}Y\texttt{<}a_{\pi(1)}, \ldots, a_{\pi(n)}\texttt{>} \in \mathbf{grArg}(\, X\texttt{<}a_1, \ldots, a_n\texttt{>} \,)$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- π is a permutation

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

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

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

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

Fig. 1. Java 5.0 type unification reduce rules with wildcards

The *reduce1* rule follows from the construction of the subtyping relation $\leq^{*}$, where from $C\texttt{<}a_1, \ldots, a_n\texttt{>} < D\texttt{<}a_{\pi(1)}, \ldots, a_{\pi(n)}\texttt{>}$ follows $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^{*} D\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ 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.

$$\text{(erase1)}\ \frac{Eq \cup \{\, \theta \lessdot \theta' \,\}}{Eq}\ \theta \leq^* \theta' \qquad \text{(erase2)}\ \frac{Eq \cup \{\, \theta \lessdot_? \theta' \,\}}{Eq}\ \theta' \in \mathbf{grArg}(\,\theta\,)$$

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

Fig. 2. Java 5.0 type unification rules with wildcards

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

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

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

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

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

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

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

where 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. 3. Java 5.0 type unification adapt rules

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 terms pairs, such that type variables are mapped to type terms, not vice versa.

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

$$\texttt{class}\ C\texttt{<}a_1, \ldots, a_n\texttt{>}\ \texttt{extends}\ D\texttt{<}D_1\texttt{<}\ldots\texttt{>}, \ldots, D_m\texttt{<}\ldots\texttt{>>}.$$

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.

Now we give an example for the type unification algorithm.

Example 9. In this example we use the standard Java 5.0 types `Number`, `Integer`, `Vector`, and `Stack`. It holds $\texttt{Integer} < \texttt{Number}$ and $\texttt{Stack<a>} < \texttt{Vector<a>}$.
As a start configuration we use

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

In step 1 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:

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

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

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

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.*

As step 2 and 3 of the type unification algorithm are the only possibilty, where the number of unifiers are multiplied, we can according to lemma 1 and theorem 1 conclude as followed.

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

Remark 2. If we consider infinite chains caused by pairs in the *extends* relation, where the number of type variables in the subtype is greater than in the supertype (cp. section 3.1), we see that all infinite subtypes are instances of a finite number of subtypes. This means that for $a \lessdot ty$, where a is a type variable and ty is an arbitrary type, there is an finite number of general unifiers, such that all other unifiers are instances of them. For example for $\{\, \mathtt{x} \lessdot \mathtt{List<a>} \,\}$ with $\mathtt{myLi<b, a>} < \mathtt{List<a>}$ there is one general unifier $\mathtt{x} \mapsto \mathtt{myLi<b, a>}$. For all other unifiers there are substitutions σ such that $\mathtt{x} \mapsto \sigma(\,\mathtt{myLi<b, a>}\,)$.

Corollary 1 means that the open problem of [5] 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 [5].
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 [8] without wildcards by wildcards, which means that we have to substitute the Java 5.0 type unification algorithm without wildcards [7] 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. Milner, R.: The definition of Standard ML (Revised). MIT Press, Cambridge, Mass. (1997)
4. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of ACM **12(1)** (January 1965) 23–41
5. Smolka, G.: Logic Programming over Polymorphically Order-Sorted Types. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany (May 1989)
6. Plümicke, M.: OBJ–P The Polymorphic Extension of OBJ–3. PhD thesis, University of Tuebingen, WSI-99-4 (1999)
7. Plümicke, M.: Type unification in Generic–Java. In Kohlhase, M., ed.: Proceedings of 18th International Workshop on Unification (UNIF'04). (July 2004)
8. 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

9. Plümicke, M.: Polymorphism in OBJ–P. In: Perspectives of System Informatics, Proceedings. Volume LNCS 1755 of Lecture Notes of Computer Science., Springer–Verlag (1999) 148–153
10. Beierle, C.: Type inferencing for polymorphic order-sorted logic programs. In: International Conference on Logic Programming. (1995) 765–779
11. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems **4** (1982) 258–282
12. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–378