

# 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 [2, 3]. 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 [4], 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 [5–8] and for the functional object-oriented language OBJ-P [9]. But in all approaches the type unification problem was not solved completely.

In [10] 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.

In this paper we extend our algorithm to type terms with wildcards and prove its soundness and completeness. 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, prove its soundness and completeness, 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 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  $a|_{ty}$ .

*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:  $TS^{(a|_{Object})} = \{A, B, I, J\}$ ,  $TS^{(a|_{I<b>} b|_{Object})} = \{C\}$ , and  $TS^{(a|_{B<a> \& J<b>} b|_{Object})} = \{D\}$ .

As  $A, I \in TS^{(a|_{Object})}$  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. `J<Integer>` is no subtype of `I<Integer>` such that 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  $C\langle\theta_1, \dots, \theta_n\rangle$  is denoted by  $CC(C\langle\theta_1, \dots, \theta_n\rangle)$ .

The inheritance hierarchy consists of two different relations: The “extends relation” (denoted by  $<$ ) 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.

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  $S\text{Type}_{TS}(BTV)$ ).** *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  $\sigma_1, \sigma_2 : BTV \rightarrow S\text{Type}_{TS}(BTV)$ , where for each type variable  $a$  of  $\theta_2$  holds  $\sigma_1(a) = \sigma_2(a)$  (soundness condition).
- $a|_{\theta_1 \& \dots \& \theta_n} \leq^* \theta_i$  for  $a \in BTV$  and  $1 \leq i \leq n$
- It holds  $C\langle\theta_1, \dots, \theta_n\rangle \leq^* C\langle\theta'_1, \dots, \theta'_n\rangle$  if for each  $\theta_i$  and  $\theta'_i$ , respectively, one of the following conditions is valid:
  - $\theta_i = ?\bar{\theta}_i$ ,  $\theta'_i = ?\bar{\theta}'_i$  and  $\bar{\theta}_i \leq^* \bar{\theta}'_i$ .
  - $\theta_i = ?\bar{\theta}_i$ ,  $\theta'_i = ?\bar{\theta}'_i$  and  $\bar{\theta}'_i \leq^* \bar{\theta}_i$ .
  - $\theta_i, \theta'_i \in S\text{Type}_{TS}(BTV)$  and  $\theta_i = \theta'_i$
  - $\theta'_i = ?\theta_i$
  - $\theta'_i = ?\theta_i$  (cp. [1] §4.5.1.1 type argument containment)
- Let  $C\langle\bar{\theta}_1, \dots, \bar{\theta}_n\rangle$  be the capture conversions of  $C\langle\theta_1, \dots, \theta_n\rangle$  and  $C\langle\bar{\theta}_1, \dots, \bar{\theta}_n\rangle \leq^* C\langle\theta'_1, \dots, \theta'_n\rangle$  then holds  $C\langle\theta_1, \dots, \theta_n\rangle \leq^* C\langle\theta'_1, \dots, \theta'_n\rangle$ .

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:

- $A\langle a \rangle \leq^* I\langle a \rangle$ , as  $A\langle a \rangle < I\langle a \rangle$
- $A\langle \text{Integer} \rangle \leq^* I\langle \text{Integer} \rangle$ , where  $\sigma_1 = [a \mapsto \text{Integer}] = \sigma_2$
- $A\langle \text{Integer} \rangle \leq^* I\langle ? \text{ extends Object} \rangle$ , as  $\text{Integer} \leq^* \text{Object}$
- $A\langle \text{Object} \rangle \leq^* I\langle ? \text{ super Integer} \rangle$ , as  $\text{Integer} \leq^* \text{Object}$

There are elements of the extends relation, where the sub-terms of a type term are not variables, like `Matrix<a> extends Vector<Vector<a>>`. As elements like this must be handled especially during the unification (*adapt* rules, Fig. 2), 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  $C\langle a_1, \dots, a_n \rangle \leq^* D\langle \theta_1, \dots, \theta_m \rangle$ , 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

$$\mathbf{Vector}\langle \mathbf{Vector}\langle a \rangle \rangle \not\leq^* \mathbf{Vector}\langle \mathbf{List}\langle a \rangle \rangle,$$

but

$$\mathbf{Vector}\langle \mathbf{Vector}\langle a \rangle \rangle \leq^* \mathbf{Vector}\langle ? \text{ extends } \mathbf{List}\langle a \rangle \rangle.$$

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

$$\begin{aligned} & \{ \mathbf{Vector}\langle a \rangle \leq^* \mathbf{AbstractList}\langle a \rangle \leq^* \mathbf{List}\langle a \rangle \\ & \mathbf{Matrix}\langle a \rangle \leq^* \mathbf{Vector}\langle \mathbf{Vector}\langle a \rangle \rangle \leq^* \mathbf{AbstractList}\langle \mathbf{Vector}\langle a \rangle \rangle \\ & \leq^* \mathbf{List}\langle \mathbf{Vector}\langle a \rangle \rangle \}. \end{aligned}$$

Now we extend the set of simple types and the corresponding subtyping relation by wildcard types. Wildcard types cannot be used, explicitly, in Java 5.0 programs. But they are allowed as instances of type variables, which means that types like this occur implicitly during the type check of Java 5.0 programs (cf. Example 4).

**Definition 3 (Extended simple types).** Let  $\mathbf{SType}_{TS}(BTV)$  be a set of simple types. The corresponding set of extended simple types is given as

$$\begin{aligned} \mathbf{ExtSType}_{TS}(BTV) = & \mathbf{SType}_{TS}(BTV) \\ & \cup \{ ? \} \\ & \cup \{ ? \text{ extends } \theta \mid \theta \in \mathbf{SType}_{TS}(BTV) \} \\ & \cup \{ ? \text{ super } \theta \mid \theta \in \mathbf{SType}_{TS}(BTV) \} \end{aligned} .$$

According to this we extend the subtyping relation  $\leq^*$  to extended simple types.

**Definition 4 (Subtyping relation  $\leq^*$  on  $\mathbf{ExtSType}_{TS}(BTV)$ ).** Let  $\leq^*$  be a subtyping relation on a given set of simple types  $\mathbf{SType}_{TS}(BTV)$ . Then  $\leq^*$  is continued on the corresponding set of extended simple types  $\mathbf{ExtSType}_{TS}(BTV)$  by: For  $\theta \leq^* \theta'$  holds  $\theta \leq^* ?\theta'$ ,  $?\theta \leq^* \theta'$ , and  $?\theta \leq^* ?\theta'$ .

*Example 4.* Let us consider the class `Vector` with its methods `addElement` and `elementAt`, respectively and two classes `Sub` and `Super` with  $\text{Sub} \leq^* \text{Super}$ . Let the following lines of Java 5.0 code be given:

```
Vector<? extends Super> v = new Vector<Sub> ();
Super sup = v.elementAt(0);
```

The type of the expression `v.elementAt(0)` is `? extends Super` and it holds  $? \text{ extends Super} \leq^* \text{Super}$ . Vice versa for

```
Vector<? super Super> v2 = new Vector<Super> ();
```

the methodcall `v2.addElement(new Sub());` is correct as  $\text{Sub} \leq^* ? \text{Super}$ . But

```
Super sup = v2.elementAt(0); //not really correct
```

is not correct, as  $? \text{Super} \not\leq^* \text{Super}$ . Furthermore, the methodcall

```
v2.addElement(v.elementAt(0));
```

is correct, as  $? \text{Super} \leq^* ? \text{Super}$  holds.

### 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]. In the following we denote  $\theta \leq \theta'$  for two type terms, which should be type unified.

#### 3.1 Overview

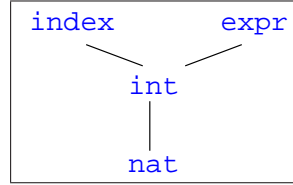
First we give an overview of approaches considering the type unification problem on polymorphically order-sorted types. In [5] the type unification problem is mentioned as an open problem. For the logical language TEL in [5] an incomplete type inference algorithm is given. The incompleteness is caused by the fact, that subtype relationships of polymorphic types which have different arities (e.g.  $\text{List}(\mathbf{a}) < \text{myLi}(\mathbf{a}, \mathbf{b})$ ) are allowed. This leads to the property that there are infinite chains in the type term ordering. In TEL for  $\text{List}(\mathbf{a}) \leq \text{myLi}(\mathbf{a}, \mathbf{b})$  holds:

$$\text{List}(\mathbf{a}) \leq \text{myLi}(\mathbf{a}, \text{List}(\mathbf{a})) \leq \text{myLi}(\mathbf{a}, \text{myLi}(\mathbf{a}, \text{List}(\mathbf{a}))) \leq \dots$$

Nevertheless, the type unification fails in some cases without infinite chains, although there is a unifier. Let for example  $\text{nat} < \text{int}$ , and the set of inequations

$\{\text{nat} \leq a, \text{int} \leq a\}$  be given, then  $\{a \mapsto \text{nat}\}$  is determined, such that  $\{\text{int} \leq \text{nat}\}$  fails, although  $\{a \mapsto \text{int}\}$  is a unifier. For  $\{\text{int} \leq a, \text{nat} \leq a\}$  the algorithm determines the correct unifier  $\{a \mapsto \text{int}\}$ .

In the typed logic programs of [7] 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 is in general no mgtu in the sense of [7]. For example for  $\text{nat} < \text{int}$ ,  $\text{neg} < \text{int}$ , and the set of inequations  $\{\text{nat} \leq a, \text{neg} \leq a\}$  the mgtu  $\{a \mapsto \text{int}\}$  is determined. If the type term ordering is extended by  $\text{int} < \text{index}$  and  $\text{int} < \text{expr}$  (cp. Fig. 1), then there are



**Fig. 1.** Simple type ordering

three unifiers  $\{a \mapsto \text{int}\}$ ,  $\{a \mapsto \text{index}\}$ , and  $\{a \mapsto \text{expr}\}$ , but none of them is a mgtu in the sense of [7].

The type system of PROTONS-L [8] was derived from TEL by disallowing any explicit subtype relationships between polymorphic type constructors.

In [8] a complete type unification algorithm is given, which can be extended to the type system of [7]. They solved the type unification problem for type term orderings following the restrictions of PROTONS-L respectively the restrictions of [7]. 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  $\text{nat} < \text{int}$ ,  $\text{neg} < \text{int}$ ,  $\text{int} < \text{index}$ , and  $\text{int} < \text{expr}$  and the set of inequations  $\{\text{nat} \leq a, \text{neg} \leq a\}$  is given, the three general unifiers  $\{a \mapsto \text{int}\}$ ,  $\{a \mapsto \text{index}\}$ , and  $\{a \mapsto \text{expr}\}$ .

Finally, in [10] we disallowed wildcards in Java 5.0 type terms, 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  $\text{myLi} \langle b, a \rangle < \text{List} \langle a \rangle$  and the set of inequations  $\{\text{myLi} \langle \text{Integer}, a \rangle \leq \text{List} \langle \text{Boolean} \rangle\}$  the general unifier  $\{a \mapsto \text{Boolean}\}$  is determined. For  $\{\text{myLi} \langle \text{Integer}, \text{Integer} \rangle \leq \text{List} \langle \text{Number} \rangle\}$  the algorithm fails, as  $\text{Integer}$  is indeed a subtype of  $\text{Number}$ , 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 again the following example: In TEL for  $\text{List}(a) \leq \text{myLi}(a, b)$  holds:

$$\text{List}(a) \leq \text{myLi}(a, \text{List}(a)) \leq \text{myLi}(a, \text{myLi}(a, \text{List}(a))) \leq \dots$$

In contrast in Java 5.0 for  $\text{myLi}(b, a) < \text{List}(a)$  holds:

$$\dots \leq^* \text{myLi}(\text{myLi}(\text{List}(a), a), a) \leq^* \text{myLi}(\text{List}(a), a) \leq^* \text{List}(a)$$

The open type unification problem of [5] 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 [11] solving the original untyped unification problem. The main difference is, that in the original unification a unifier  $\sigma$  is demanded, such that  $\sigma(\theta_1) = \sigma(\theta_2)$ , whereas in our case a 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  $\{[a \mapsto \bar{\theta}] \mid \bar{\theta} \leq^* \theta\}$  in our algorithm.

### 3.2 Type unification algorithm

In the description of the algorithm we denote also  $\theta \leq \theta'$  for two type terms, which should be type unified. During the unification algorithm  $\leq$  is replaced by  $\leq_?$  and  $\doteq$ , respectively.  $\theta \leq_? \theta'$  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  $\sigma(\theta) = \sigma(\theta')$ .

The next definition gives a form of the set of equations for the end configuration of the algorithms.

**Definition 5. (Solved form)** A set of equations  $Eq$  is in *solved form*, if

$$Eq = \{a_1 \doteq \theta_1, \dots, a_n \doteq \theta_n\}$$

where  $a_1, \dots, a_n \in BTV$  are pairwise different bounded type variables,  $\theta_1, \dots, \theta_n \in \text{ExtSType}_{TS}(BTV)$  are extended simple types, and for all  $i, j \in \{1, \dots, n\}$  holds:  $a_i$  does not occur in  $\theta_j$ .

In the following, the type unification algorithm is shown. The type unification algorithm is given as follows

**Input:** Set of equations  $Eq = \{\theta_1 \leq \theta'_1, \dots, \theta_n \leq \theta'_n\}$

**Precondition:**  $\theta_i, \theta'_i \in \text{SType}_{TS}(BTV)$  for  $1 \leq i \leq n$ .

**Output:** Set of all general type unifiers  $Uni = \{\sigma_1, \dots, \sigma_m\}$

**Postcondition:** For all  $1 \leq j \leq m$  and for all  $1 \leq i \leq n$  holds  $(\sigma_j(\theta_i) \leq^* \sigma_j(\theta'_i))$ .

(adapt)	$\frac{Eq \cup \{ D \langle \theta_1, \dots, \theta_n \rangle \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}{Eq \cup \{ D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n] \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}$ <p>where there are <math>\bar{\theta}'_1, \dots, \bar{\theta}'_m</math> with</p> <p>– <math>(D \langle a_1, \dots, a_n \rangle \leq^* D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle) \in \mathbf{FC}(&lt;)</math></p>
(adaptExt)	$\frac{Eq \cup \{ D \langle \theta_1, \dots, \theta_n \rangle \leq ? D' \langle \theta'_1, \dots, \theta'_m \rangle \}}{Eq \cup \{ D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n] \leq ? D' \langle \theta'_1, \dots, \theta'_m \rangle \}}$ <p>where <math>D \in \Theta^{(n)}</math> or <math>D = ?\bar{X}</math> with <math>\bar{X} \in \Theta^{(n)}</math> and there are <math>\bar{\theta}'_1, \dots, \bar{\theta}'_m</math> with</p> <p>– <math>?D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle \in \mathbf{grArg}(D \langle a_1, \dots, a_n \rangle)</math></p>
(adaptSup)	$\frac{Eq \cup \{ D' \langle \theta'_1, \dots, \theta'_m \rangle \leq ? D \langle \theta_1, \dots, \theta_n \rangle \}}{Eq \cup \{ D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n] \leq ? D' \langle \theta'_1, \dots, \theta'_m \rangle \}}$ <p>where <math>D' \in \Theta^{(n)}</math> or <math>D' = ?\bar{X}</math> with <math>\bar{X} \in \Theta^{(n)}</math> and there are <math>\bar{\theta}'_1, \dots, \bar{\theta}'_m</math> with</p> <p>– <math>?D \langle a_1, \dots, a_n \rangle \in \mathbf{grArg}(D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle)</math></p>

**Fig. 2.** Java 5.0 type unification adapt rules

The algorithm itself is given in seven steps:

1. Repeated application of the *adapt* rules (Fig. 2), the *reduce* rules, the *erase* rules and the *swap* rule (Fig. 3) to all elements of  $Eq$ . The end configuration of  $Eq$  is reached if for each element no rule is applicable.
2.  $Eq'_1 = \text{Subset of pairs, where both type terms are type variables}$
3.  $Eq'_2 = Eq \setminus Eq'_1$
4.  $Eq'_{set}$ 

$$= \{ Eq'_1 \} \times \left( \bigotimes_{(a \leq \theta') \in Eq'_2} \{ ([a \doteq \theta] \cup \sigma) \mid \begin{array}{l} (\bar{\theta} \leq^* C \langle \theta_1, \dots, \theta_n \rangle) \in \mathbf{FC}(<), \\ \bar{\theta}' \in \{ C \langle \theta'_1, \dots, \theta'_n \rangle \\ \mid \theta'_i \in \mathbf{grArg}(\theta_i), 1 \leq i \leq n \} \\ \sigma \in \mathbf{Unify}(\bar{\theta}', \theta'), \\ \theta \in \mathbf{smaller}(\sigma(\bar{\theta})) \} \} \right) \\ \times \left( \bigotimes_{(a \leq ? \theta') \in Eq'_2} \{ ([a \doteq \theta] \cup \sigma) \mid \begin{array}{l} (\bar{\theta} \leq^* C \langle \theta_1, \dots, \theta_n \rangle) \in \mathbf{FC}(<), \\ \bar{\theta}' \in \{ C \langle \theta'_1, \dots, \theta'_n \rangle \\ \mid \theta'_i \in \mathbf{grArg}(\theta_i), 1 \leq i \leq n \} \\ \sigma \in \mathbf{Unify}(\bar{\theta}', \theta'), \\ \theta \in \mathbf{smArg}(\sigma(? \bar{\theta})) \} \} \right)$$



$$\begin{array}{l}
\text{(reduceUp)} \quad \frac{Eq \cup \{ \theta \leq ? \theta' \}}{Eq \cup \{ \theta \leq \theta' \}} \quad \text{(reduceUpLow)} \quad \frac{Eq \cup \{ ? \theta \leq ? \theta' \}}{Eq \cup \{ \theta \leq \theta' \}} \\
\\
\text{(reduceLow)} \quad \frac{Eq \cup \{ ? \theta \leq \theta' \}}{Eq \cup \{ \theta \leq \theta' \}} \\
\\
\text{(reduce1)} \quad \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \leq D \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \leq ? \theta'_1, \dots, \theta_{\pi(n)} \leq ? \theta'_n \}} \\
\text{where} \\
- C \langle a_1, \dots, a_n \rangle \leq^* D \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \\
- \{ a_1, \dots, a_n \} \subseteq BTV \\
- \pi \text{ is a permutation} \\
\\
\text{(reduceExt)} \quad \frac{Eq \cup \{ X \langle \theta_1, \dots, \theta_n \rangle \leq ? Y \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \leq ? \theta'_1, \dots, \theta_{\pi(n)} \leq ? \theta'_n \}} \\
\text{where} \\
- ? Y \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \in \mathbf{grArg}(X \langle a_1, \dots, a_n \rangle) \\
- \{ a_1, \dots, a_n \} \subseteq BTV \\
- \pi \text{ is a permutation} \\
- X \in \Theta^{(n)} \text{ or } X = ? \bar{X} \text{ with } \bar{X} \in \Theta^{(n)}. \\
\\
\text{(reduceSup)} \quad \frac{Eq \cup \{ X \langle \theta_1, \dots, \theta_n \rangle \leq ? Y \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta'_1 \leq ? \theta_{\pi(1)}, \dots, \theta'_n \leq ? \theta_{\pi(n)} \}} \\
\text{where} \\
- ? Y \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \in \mathbf{grArg}(X \langle a_1, \dots, a_n \rangle) \\
- \{ a_1, \dots, a_n \} \subseteq BTV \\
- \pi \text{ is a permutation} \\
- X \in \Theta^{(n)} \text{ or } X = ? \bar{X} \text{ with } \bar{X} \in \Theta^{(n)}. \\
\\
\text{(reduceEq)} \quad \frac{Eq \cup \{ X \langle \theta_1, \dots, \theta_n \rangle \leq ? X \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \doteq \theta'_1, \dots, \theta_{\pi(n)} \doteq \theta'_n \}} \\
\\
\text{(reduce2)} \quad \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \doteq C \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_1 \doteq \theta'_1, \dots, \theta_n \doteq \theta'_n \}} \\
\text{where} \\
- C \in \Theta^{(n)} \text{ or } C = ? \bar{C} \text{ or } C = ? \overline{C} \text{ with } \bar{C} \in \Theta^{(n)}, \text{ respectively.} \\
\\
\text{(erase1)} \quad \frac{Eq \cup \{ \theta \leq \theta' \}}{Eq} \theta \leq^* \theta' \quad \text{(erase2)} \quad \frac{Eq \cup \{ \theta \leq ? \theta' \}}{Eq} \theta' \in \mathbf{grArg}(\theta) \\
\\
\text{(erase3)} \quad \frac{Eq \cup \{ \theta \doteq \theta' \}}{Eq} \theta = \theta' \quad \text{(swap)} \quad \frac{Eq \cup \{ \theta \doteq a \}}{Eq \cup \{ a \doteq \theta \}} \theta \notin BTV, a \in BTV
\end{array}$$

**Fig. 3.** Java 5.0 type unification rules with wildcards

$$\begin{aligned}
& \times ( \bigotimes_{(a \leq_? \theta') \in Eq'_2} \{ [a \doteq \theta'] \mid \theta' \in \mathbf{smArg}(^? \theta') \} ) \\
& \times ( \bigotimes_{(a \leq_? \theta') \in Eq'_2} \{ [a \doteq \theta'] \} ) \\
& \times ( \bigotimes_{(\theta \leq a) \in Eq'_2} \{ [a \doteq \theta'] \mid \theta' \in \mathbf{greater}(\theta) \} ) \\
& \times ( \bigotimes_{(^? \theta \leq_? a) \in Eq'_2} \{ [a \doteq \theta'] \mid \theta' \in \mathbf{grArg}(^? \theta) \} ) \\
& \times ( \bigotimes_{(^? \theta \leq_? a) \in Eq'_2} \{ ([a \doteq ^? \theta'] \cup \sigma) \mid (\bar{\theta} \leq^* C \langle \theta_1, \dots, \theta_n \rangle) \in \mathbf{FC}(<), \\
& \quad \bar{\theta}' \in \{ C \langle \theta'_1, \dots, \theta'_n \rangle \\
& \quad \mid \theta'_i \in \mathbf{grArg}(\theta_i), 1 \leq i \leq n \} \\
& \quad \sigma \in \mathbf{Unify}(\bar{\theta}', \theta), \\
& \quad \theta' \in \mathbf{smaller}(\sigma(\bar{\theta})) \} ) \\
& \times ( \bigotimes_{(\theta \leq_? a) \in Eq'_2} \{ [a \doteq \theta'] \mid \theta' \in \mathbf{grArg}(\theta) \} ) \\
& \times \{ [a \doteq \theta \mid (a \doteq \theta) \in Eq'_2] \}
\end{aligned}$$

5. Application of the following *subst* rule

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

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.  $Uni = \{ \sigma \mid Eq'' \in Eq''_{set}, Eq'' \text{ is in solved form},$   
 $\sigma = \{ a \mapsto \theta \mid (a \doteq \theta) \in Eq'' \} \}$

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**. These functions determine all supertypes, respectively all 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. 2 and 3.

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

`class  $C \langle a_1, \dots, a_n \rangle$  extends  $D \langle D_1 \langle \dots \rangle, \dots, D_m \langle \dots \rangle \rangle$ .`

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 (Def. 4).

The *reduce1* rule follows from the construction of the subtyping relation  $\leq^*$ , where  $C(a_1, \dots, a_n) < D(a_{\pi(1)}, \dots, a_{\pi(n)})$  leads to  $C(\theta_1, \dots, \theta_n) \leq^* D(\theta'_1, \dots, \theta'_n)$  if and only if  $\theta'_i \in \mathbf{grArg}(\theta_{\pi(i)})$  for  $1 \leq i \leq n$ .

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, Def. 1).

**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 5.* In this example we use the standard Java 5.0 types `Number`, `Integer`, `Stack`, `Vector`, `AbstractList`, and `List`. It holds  $\text{Integer} < \text{Number}$  and  $\text{Stack}\langle a \rangle < \text{Vector}\langle a \rangle < \text{AbstractList}\langle a \rangle < \text{List}\langle a \rangle$ .

As a start configuration we use

$\{ (\text{Stack}\langle a \rangle < \text{Vector}\langle ?\text{Number} \rangle), (\text{AbstractList}\langle \text{Integer} \rangle < \text{List}\langle a \rangle) \}$ .

In the first step the *reduce1* rule is applied twice:

$$\{ a < ?\text{Number}, \text{Integer} < ?a \}$$

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

$$\begin{aligned} & \{ \{ a \doteq ?\text{Number}, a \doteq \text{Integer} \}, \{ a \doteq ?\text{Number}, a \doteq ?\text{Number} \}, \\ & \{ a \doteq ?\text{Number}, a \doteq ?\text{Integer} \}, \{ a \doteq ?\text{Number}, a \doteq ?\text{Integer} \}, \\ & \{ a \doteq \text{Number}, a \doteq \text{Integer} \}, \{ a \doteq \text{Number}, a \doteq ?\text{Number} \}, \\ & \{ a \doteq \text{Number}, a \doteq ?\text{Integer} \}, \{ a \doteq \text{Number}, a \doteq ?\text{Integer} \}, \\ & \{ a \doteq ?\text{Integer}, a \doteq \text{Integer} \}, \{ a \doteq ?\text{Integer}, a \doteq ?\text{Number} \}, \\ & \{ a \doteq ?\text{Integer}, a \doteq ?\text{Integer} \}, \{ a \doteq ?\text{Integer}, a \doteq ?\text{Integer} \}, \\ & \{ a \doteq \text{Integer}, a \doteq \text{Integer} \}, \{ a \doteq \text{Integer}, a \doteq ?\text{Number} \}, \\ & \{ a \doteq \text{Integer}, a \doteq ?\text{Integer} \} \{ a \doteq \text{Integer}, a \doteq ?\text{Integer} \} \} \end{aligned}$$

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

$$\begin{aligned} & \{ \{ \text{Integer} \doteq ?\text{Number}, a \doteq \text{Integer} \}, \{ ?\text{Number} \doteq ?\text{Number}, a \doteq ?\text{Number} \}, \\ & \{ ?\text{Integer} \doteq ?\text{Number}, a \doteq ?\text{Integer} \}, \{ ?\text{Integer} \doteq ?\text{Number}, a \doteq ?\text{Integer} \}, \\ & \{ \text{Integer} \doteq \text{Number}, a \doteq \text{Integer} \}, \{ ?\text{Number} \doteq \text{Number}, a \doteq ?\text{Number} \}, \\ & \{ ?\text{Integer} \doteq \text{Number}, a \doteq ?\text{Integer} \}, \{ ?\text{Integer} \doteq \text{Number}, a \doteq ?\text{Integer} \}, \\ & \{ \text{Integer} \doteq ?\text{Integer}, a \doteq \text{Integer} \}, \{ ?\text{Number} \doteq ?\text{Integer}, a \doteq ?\text{Number} \}, \\ & \{ ?\text{Integer} \doteq ?\text{Integer}, a \doteq ?\text{Integer} \}, \{ ?\text{Integer} \doteq ?\text{Integer}, a \doteq ?\text{Integer} \}, \\ & \{ \text{Integer} \doteq \text{Integer}, a \doteq \text{Integer} \}, \{ ?\text{Number} \doteq \text{Integer}, a \doteq ?\text{Number} \}, \\ & \{ ?\text{Integer} \doteq \text{Integer}, a \doteq ?\text{Integer} \} \{ ?\text{Integer} \doteq \text{Integer}, a \doteq ?\text{Integer} \} \} \end{aligned}$$

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 three general type unifiers:

$$\{ \{ a \mapsto ?\text{Number} \}, \{ a \mapsto ?\text{Integer} \}, \{ a \mapsto \text{Integer} \} \}.$$

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 6.* Let  $\text{myLi}\langle b, a \rangle < \text{List}\langle a \rangle$  and the start configuration  $\{ \text{List}\langle x \rangle < \text{List}\langle ?\text{List}\langle \text{Integer} \rangle \rangle \}$  be given. In the first step the reduce 1 rule is applied:  $\{ x < ?\text{List}\langle \text{Integer} \rangle \}$ . With the fourth step we get the result:

$$\begin{aligned} & \{ \{ x \mapsto \text{List}\langle \text{Integer} \rangle \}, \{ x \mapsto ?\text{List}\langle \text{Integer} \rangle \}, \\ & \{ x \mapsto \text{myLi}\langle b, \text{Integer} \rangle \}, \{ x \mapsto ?\text{myLi}\langle b, \text{Integer} \rangle \} \}. \end{aligned}$$

All other infinite numbers of unifiers are instances of these general unifiers (e.g.  $\{ x \mapsto ?\text{myLi}\langle ?\text{List}\langle \text{Integer} \rangle, \text{Integer} \rangle \}$  or  $\{ x \mapsto ?\text{myLi}\langle ?\text{myList}\langle b, \text{Integer} \rangle, \text{Integer} \rangle \}$ ).

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.*

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

### Soundness

We have to prove that each calculated result of the algorithm is a general unifier of the corresponding input. We do the proof as follows: For an arbitrary result  $\sigma$  we show that if  $\sigma$  is a general unifier of the set of type terms after a transformation, then  $\sigma$  is also a general unifier of the set of type term pairs before the transformation.

Let  $\sigma = \{ a_1 \mapsto \theta_1, \dots, a_n \mapsto \theta_n \}$  be a result of the algorithm.

**Step seven:** In step seven the equations  $a_i \doteq \theta_i$  are transformed to the mappings  $a_i \mapsto \theta_i$ . It is obvious that  $\sigma$  is a general unifier of

$$\{ a_1 \doteq \theta_1, \dots, a_n \doteq \theta_n \}.$$

**Step six:** In step six different sets of type term pairs are united. This means that  $\sigma$  is also a general unifier before step six is applied.

**Step five:** The type variables  $a$  are substituted by  $\theta$ . As for  $a \doteq \theta$  holds  $\sigma(a) = \sigma(\theta)$ ,  $\sigma$  is also a general unifier of the set of type terms before the substitution.

**Step four:** We have to consider eight different cases in step four.

$(a < \theta') \in Eq'_2$ : In this case type term pairs of the form  $(a < \theta')$  are transformed to  $[(a \doteq \theta) \cup \sigma']$  for a type term  $\theta$  with  $\theta \in \mathbf{smaller}(\sigma'(\theta'))$ .

If  $\sigma$  is a general unifier of  $[(a \doteq \theta) \cup \sigma']$  then  $\sigma$  is also a general unifier of  $(a < \theta')$  for  $\theta \in \mathbf{smaller}(\sigma'(\theta'))$ .

$(a <_{?} \theta') \in Eq'_2$ : This case is analogous to the first case.

$(a <_{?} \theta') \in Eq'_2$ : In this case type term pairs of the form  $(a <_{?} \theta')$  are transformed to  $a \doteq \theta'$  for a type term  $\theta'$  with  $\theta' \in \mathbf{smArg}(\theta')$ .

If  $\sigma$  is a general unifier of  $(a \doteq \theta')$  then  $\sigma$  is also a general unifier of  $(a <_{?} \theta')$  for  $\theta' \in \mathbf{smArg}(\theta')$ .

$(a <_{?} \theta') \in Eq'_2$ : It is obvious, that a general unifier of  $(a \doteq \theta')$  is also a general unifier of  $(a <_{?} \theta')$ .

$(\theta < a) \in Eq'_2, (\theta <_{?} a) \in Eq'_2$ : The proof of these cases is analogous to the proof of case  $(a <_{?} \theta') \in Eq'_2$ .

$(\theta <_{?} a) \in Eq'_2$ : The proof of this case is analogous to the proof of case  $(a < \theta') \in Eq'_2$ .

$(\theta <_{?} a) \in Eq'_2$ : The proof of this case is analogous to the proof of case  $(a <_{?} \theta') \in Eq'_2$ .

**Step two and three:** In these steps the pairs are only filtered. This means that the pairs are unchanged. But this means that  $\sigma$  is also a general unifier before applying the steps.

**Step one:** The soundness of the *reduce2* rule, the *erase3* rule, and the *swap* rule follows directly from the original unification algorithm of [11] as they are unchanged.

The soundness of the *erase1* rule and the *erase2* rule are obvious.

The soundness of the *reduceUp* rule, the *reduceUpLow* rule, and the *reduceLow* rule follows directly from the definition of the subtyping ordering on  $\text{ExtSType}_{TS}(BT\mathcal{V})$  (Def. 4), as the following relationships are equivalent  $\theta \leq^* \theta', \theta \leq_{?} \theta', \theta \leq^* \theta',$  and  $\theta \leq_{?} \theta'$ .

**reduce1 rule:** If  $\sigma$  is a general unifier of  $\theta_{\pi(i)} <_{?} \theta'_i$  for  $1 \leq i \leq n$ , it holds  $\sigma(\theta'_i) \in \mathbf{grArg}(\sigma(\theta_{\pi(i)}))$  for  $1 \leq i \leq n$ . With

$$C \langle a_1, \dots, a_n \rangle \leq^* D \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle$$

follows also by Def. 1:

$$\sigma(C \langle \theta_1, \dots, \theta_n \rangle) \leq^* \sigma(D \langle \theta'_1, \dots, \theta'_n \rangle)$$

**reduceExt rule:** If  $\sigma$  is a general unifier of  $\theta_{\pi(i)} <_{?} \theta'_i$  for  $1 \leq i \leq n$ , it holds  $\sigma(\theta'_i) \in \mathbf{grArg}(\sigma(\theta_{\pi(i)}))$  for  $1 \leq i \leq n$ . With

$$Y \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \in \mathbf{grArg}(X \langle a_1, \dots, a_n \rangle)$$

follows also by Def. 1:

$$\sigma(Y \langle \theta'_1, \dots, \theta'_n \rangle) \in \mathbf{grArg}(\sigma(X \langle \theta_1, \dots, \theta_n \rangle)),$$

which means that  $\sigma$  is a general of  $X \langle \theta_1, \dots, \theta_n \rangle <_{?} Y \langle \theta'_1, \dots, \theta'_n \rangle$ .

**reduceSup rule:** If  $\sigma$  is a general unifier of  $\theta'_i \leq ? \theta_{\pi(i)}$  for  $1 \leq i \leq n$ , it holds  $\sigma(\theta_{\pi(i)}) \in \mathbf{grArg}(\sigma(\theta'_i))$  for  $1 \leq i \leq n$ . With

$$?Y\langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \in \mathbf{grArg}(X\langle a_1, \dots, a_n \rangle)$$

follows also by Def. 1:

$$\sigma(?Y\langle \theta'_1, \dots, \theta'_n \rangle) \in \mathbf{grArg}(\sigma(X\langle \theta_1, \dots, \theta_n \rangle)),$$

which means that  $\sigma$  is a general unifier of  $X\langle \theta_1, \dots, \theta_n \rangle \leq ? Y\langle \theta'_1, \dots, \theta'_n \rangle$ .

**reduceEq rule:** If it holds for the general unifier  $\sigma$ :  $\sigma(\theta_{\pi(i)}) = \sigma(\theta'_i)$  for  $1 \leq i \leq n$  then follows by Def. 1:

$$\sigma(X\langle \theta'_1, \dots, \theta'_n \rangle) \in \mathbf{grArg}(\sigma(X\langle \theta_1, \dots, \theta_n \rangle)),$$

which means that  $\sigma$  is a general unifier of  $X\langle \theta_1, \dots, \theta_n \rangle \leq ? X\langle \theta'_1, \dots, \theta'_n \rangle$ .

**adapt rule:** If it holds for the general unifier  $\sigma$ :

$$\sigma(D'\langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n]) \leq^* \sigma(D'\langle \theta'_1, \dots, \theta'_m \rangle),$$

and

$$D\langle a_1, \dots, a_n \rangle \leq^* D'\langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle$$

then follows also by Def. 1:

$$\sigma(D\langle \theta_1, \dots, \theta_n \rangle) \leq^* \sigma(D'\langle \theta'_1, \dots, \theta'_m \rangle).$$

**adaptExt rule:** If  $\sigma$  is a general unifier of  $D'\langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n] \leq ? D'\langle \theta'_1, \dots, \theta'_m \rangle$ , it holds:

$$\begin{aligned} &\sigma(?D'\langle \theta'_1, \dots, \theta'_m \rangle) \\ &\in \mathbf{grArg}(\sigma(D'\langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n])). \end{aligned}$$

With

$$?D'\langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle \in \mathbf{grArg}(D\langle a_1, \dots, a_n \rangle)$$

follows also by Def. 1:

$$\sigma(?D'\langle \theta'_1, \dots, \theta'_m \rangle) \in \mathbf{grArg}(\sigma(D\langle \theta_1, \dots, \theta_n \rangle)),$$

which means that  $\sigma$  is a general unifier of  $D\langle \theta_1, \dots, \theta_n \rangle \leq ? D'\langle \theta'_1, \dots, \theta'_m \rangle$ .

**adaptSup rule:** If  $\sigma$  is a general unifier of  $D'\langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n] \leq ? D'\langle \theta'_1, \dots, \theta'_m \rangle$ , it holds:

$$\begin{aligned} &\sigma(?D'\langle \theta'_1, \dots, \theta'_m \rangle) \\ &\in \mathbf{grArg}(\sigma(D'\langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n])). \end{aligned}$$

With

$$?D\langle a_1, \dots, a_n \rangle \in \mathbf{grArg}(D'\langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle)$$

follows also by Def. 1:

$$\sigma(?D\langle \theta_1, \dots, \theta_n \rangle) \in \mathbf{grArg}(\sigma(D'\langle \theta'_1, \dots, \theta'_m \rangle)),$$

which means that  $\sigma$  is a general unifier of  $D'\langle \theta'_1, \dots, \theta'_m \rangle \leq ? D\langle \theta_1, \dots, \theta_n \rangle$ .

Summarized, this means that  $\sigma$  is also a general unifier of the type term set before the application of this step.

We have proved for all transformations that if a substitution is a general unifier of the result the same substitution is also a general unifier of the input. This means that the algorithm is sound.

### Completeness

For the completeness we have to prove for an arbitrary general unifier  $\sigma$  of an input set of type term pairs, that  $\sigma$  is determined by the algorithm. We do the proof, as we show for each transformation of the algorithm, if  $\sigma$  is a general unifier of the set of type terms before a transformation is done,  $\sigma$  is also a general unifier of at least one set of type term pairs after the transformation.

Let  $\sigma$  be a general unifier of  $Eq = \{ \theta_1 < \theta'_1, \dots, \theta_n < \theta'_n \}$

**Step one:** For the *reduce2* rule, the *erase3* rule, and the *swap* rule it follows directly from the original unification algorithm in [11] that all general unifier are obtained, as the rules are unchanged.

It is obvious that the *erase1* rule and the *erase2* rule obtain all unifiers.

For the *reduceUp* rule, the *reduceUpLow* rule, and the *reduceLow* rule it follows directly from the definition of the subtyping ordering on  $\text{ExtSType}_{TS}(BTV)$  (Def. 4), that they obtain all unifiers, as the following relationships are equivalent  $\theta \leq^* \theta'$ ,  $?\theta \leq^* \theta'$ ,  $\theta \leq^* ?\theta'$ , and  $?\theta \leq^* ?\theta'$ .

**reduce1 rule:** If  $\sigma$  is a general unifier of  $C < \theta_1, \dots, \theta_n > \leq D < \theta'_1, \dots, \theta'_n >$  from  $C < a_1, \dots, a_n > \leq^* D < a_{\pi(1)}, \dots, a_{\pi(n)} >$  with Def. 1 follows  $\sigma(\theta'_i) \in \mathbf{grArg}(\sigma(\theta_{\pi(i)}))$  for  $1 \leq i \leq n$ . This means that  $\sigma$  is also a general unifier of  $\theta_{\pi(i)} < ?\theta'_i$  for  $1 \leq i \leq n$ .

**reduceExt rule:** If  $\sigma$  is a general unifier of  $X < \theta_1, \dots, \theta_n > \leq ?Y < \theta'_1, \dots, \theta'_n >$  from  $?Y < a_{\pi(1)}, \dots, a_{\pi(n)} > \in \mathbf{grArg}(X < a_1, \dots, a_n >)$  with Def. 1 follows  $\sigma(\theta'_i) \in \mathbf{grArg}(\sigma(\theta_{\pi(i)}))$  for  $1 \leq i \leq n$ . This means that  $\sigma$  is also a general unifier of  $\theta_{\pi(i)} < ?\theta'_i$  for  $1 \leq i \leq n$ .

**reduceSup rule:** If  $\sigma$  is a general unifier of  $X < \theta_1, \dots, \theta_n > \leq ?Y < \theta'_1, \dots, \theta'_n >$  from  $?Y < a_{\pi(1)}, \dots, a_{\pi(n)} > \in \mathbf{grArg}(X < a_1, \dots, a_n >)$  with Def. 1 follows  $\sigma(\theta_{\pi(i)}) \in \mathbf{grArg}(\sigma(\theta'_i))$  for  $1 \leq i \leq n$ . This means that  $\sigma$  is also a general unifier of  $\theta'_i < ?\theta_{\pi(i)}$  for  $1 \leq i \leq n$ .

**reduceEq rule:** If  $\sigma$  is a general unifier of  $X < \theta_1, \dots, \theta_n > \leq ?X < \theta'_1, \dots, \theta'_n >$  with Def. 1 follows  $\sigma(\theta_i) = \sigma(\theta'_i)$  for  $1 \leq i \leq n$ . This means that  $\sigma$  is also a general unifier of  $\theta_i \doteq \theta'_i$  for  $1 \leq i \leq n$ .

**adapt rule:** If  $\sigma$  is a general unifier of  $D < \theta_1, \dots, \theta_n > \leq D' < \theta'_1, \dots, \theta'_m >$  from  $D < a_1, \dots, a_n > \leq^* D' < \bar{\theta}'_1, \dots, \bar{\theta}'_m > \in \mathbf{FC}(\leq^*)$  with Def. 1 follows that  $\sigma$  is also a general unifier of

$$D' < \bar{\theta}'_1, \dots, \bar{\theta}'_m > [a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n] \leq D' < \theta'_1, \dots, \theta'_m >.$$

**adaptExt rule:** If  $\sigma$  is a general unifier of  $D\langle\theta_1, \dots, \theta_n\rangle \leq_? D'\langle\theta'_1, \dots, \theta'_m\rangle$  from  ${}^?D'\langle\bar{\theta}'_1, \dots, \bar{\theta}'_m\rangle \in \mathbf{grArg}(D\langle a_1, \dots, a_n\rangle)$  follows that  $\sigma$  is also a general unifier of

$$D'\langle\bar{\theta}'_1, \dots, \bar{\theta}'_m\rangle[a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n] \leq_? {}^?D'\langle\theta'_1, \dots, \theta'_m\rangle.$$

**adaptSup rule:** If  $\sigma$  is a general unifier of  $D'\langle\theta'_1, \dots, \theta'_m\rangle \leq_? D\langle\theta_1, \dots, \theta_m\rangle$  from  ${}^?D\langle a_1, \dots, a_n\rangle \in \mathbf{grArg}(D'\langle\bar{\theta}'_1, \dots, \bar{\theta}'_m\rangle)$  follows that  $\sigma$  is also a general unifier of

$$D'\langle\bar{\theta}'_1, \dots, \bar{\theta}'_m\rangle[a_i \mapsto CC(\theta_i) \mid 1 \leq i \leq n] \leq_? {}^?D'\langle\theta'_1, \dots, \theta'_m\rangle.$$

This means that after step one  $\sigma$  is still a general unifier of the transformed set of type term pairs  $Eq$ .

**Step two and three:** In these steps the pairs are only filtered. This means that the pairs are unchanged. But this means that  $\sigma$  is also a general unifier after applying the steps.

**Step four:** We have to consider eight different cases:

$(a \leq \theta') \in Eq'_2$ : It holds  $\sigma(a) \leq^* \sigma(\theta')$ . With Def. 1 follows

- there is a  $(\bar{\theta} \leq^* C\langle\theta'_1, \dots, \theta'_n\rangle) \in \mathbf{FC}(<)$
- for  $1 \leq i \leq n$ : there are  $(\theta''_i \in \mathbf{grArg}(\theta'_i))$ ,
- there is a substitution  $\sigma' = \text{Unify}(C\langle\theta''_1, \dots, \theta''_n\rangle, \theta')$ ,  
and  $\theta \in \mathbf{smaller}(\sigma'(\bar{\theta}))$  such that  $\sigma(a) = \theta$ .

This means for the pair  $(a \leq \theta') \in Eq'_2$ , where  $\sigma$  is a general unifier, there is a set of equations in  $Eq_{set}$  after the transformation with a pair  $(a \doteq \theta)$ , where  $\sigma$  is still a general unifier.

$(a \leq_? \theta') \in Eq'_2$ : This case is analogous to the first case.

$(a \leq_? \theta') \in Eq'_2$ : It holds  $\sigma({}^?\theta') \in \mathbf{grArg}(\sigma(a))$ . From this follows by Def. 1 that there is a type term  $\theta' \in \mathbf{smArg}({}^?\theta')$  and  $\sigma(a) = \sigma(\theta')$ .

This means for the pair  $(a \leq_? \theta') \in Eq'_2$ , where  $\sigma$  is a general unifier, there is a set of equations in  $Eq_{set}$  after the transformation with a pair  $(a \doteq \theta')$ , where  $\sigma$  is still a general unifier.

$(a \leq_? \theta') \in Eq'_2$ : It is obvious, that a general unifier of  $(a \leq_? \theta')$  is also a general unifier of  $(a \doteq \theta')$ .

$(\theta \leq a) \in Eq'_2$ ,  $({}^?\theta \leq_? a) \in Eq'_2$ : The proof of these cases is analogous to the proof of case  $(a \leq_? \theta') \in Eq'_2$

$({}^?\theta \leq_? a) \in Eq'_2$ : The proof of this case is analogous to the proof of case  $(a \leq \theta') \in Eq'_2$ .

$(\theta \leq_? a) \in Eq'_2$ : The proof of this case is analogous to the proof of case  $(a \leq_? \theta') \in Eq'_2$

As in step four, the Cartesian product of the respective results is built, and there is one set of type term pairs of  $Eq_{set}$ , where  $\sigma$  is still a general unifier.

**Step five:** If  $\sigma$  is a general unifier of  $\{a \doteq \theta\}$ , then holds  $\sigma(a[a \mapsto \theta]) = \sigma(\theta)$ .

From this follows that for all pairs  $(\bar{\theta} \leq \theta') \in Eq''$ ,  $(\bar{\theta} \leq_? \theta') \in Eq''$ , and  $(\bar{\theta} \doteq$



$\bar{\theta}' \in Eq''$ , respectively, with  $\sigma(\bar{\theta}) \leq^* \sigma(\bar{\theta}')$   $\sigma(\bar{\theta}') \in \mathbf{grArg}(\sigma(\bar{\theta}))$ , and  $\sigma(\bar{\theta}) = \sigma(\bar{\theta}')$ , respectively, holds  $\sigma(\bar{\theta}[a \mapsto \theta]) \leq^* \sigma(\bar{\theta}'[a \mapsto \theta])$ ,  $\sigma(\bar{\theta}'[a \mapsto \theta]) \in \mathbf{grArg}(\sigma(\bar{\theta}[a \mapsto \theta]))$ , and  $\sigma(\bar{\theta}[a \mapsto \theta]) = \sigma(\bar{\theta}'[a \mapsto \theta])$ , respectively. This means that after step five  $\sigma$  is still a general unifier of the transformed set of type term pairs  $Eq'$ .

**Step six:** As no pair of type terms is transformed, the unifiers are maintained.

**Step seven:** In step seven the pairs of type terms which are in solved form are filtered. All other sets of pairs of type terms have no unifier, thus all unifiers are obtained.

We have proved that for an arbitrary given general unifier  $\sigma$  of

$$\{\theta_1 < \theta'_1, \dots, \theta_n < \theta'_n\}$$

$\sigma$  is calculated by the algorithm. This means that the algorithm is complete.

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.*

In Example 6 is shown, how the problem of infinite chains is solved.

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

Corollary 1 means that the open problem of [5], type unification in TEL, is also solved by our type unification algorithm. In TEL there are no wildcards. This means that in the type unification algorithm the differentiation between  $<$ ,  $<?$ , and  $\doteq$  is unnecessary. Following this, the rules *reduceExt*, *reduceSup*, *reduceEq*, and *reduce2* and the functions **smArg** and **grArg** are also unnecessary. Furthermore in the finite closure there are only type terms of the form  $C(a_1, \dots, a_n)$  where the  $a_i$  are type variables, which means that the adapt rules are also unnecessary. Finally in TEL in the ordering  $<$  for  $\theta < \theta'$  holds  $\mathbf{TVar}(\theta) \subseteq \mathbf{TVar}(\theta')$ , while in Java 5.0 holds  $\mathbf{TVar}(\theta') \subseteq \mathbf{TVar}(\theta)$ . This means that infinite chains in TEL has lower bounds, while infinite chains in Java 5.0 has upper bounds. Let us consider again the example from section 3.1. In TEL for  $\mathbf{List}(a) \leq \mathbf{myLi}(a, b)$  holds:

$$\mathbf{List}(a) \leq \mathbf{myLi}(a, \mathbf{List}(a)) \leq \mathbf{myLi}(a, \mathbf{myLi}(a, \mathbf{List}(a))) \leq \dots$$

In contrast in Java 5.0 for  $\mathbf{myLi}(b, a) < \mathbf{List}(a)$  holds:

$$\dots \leq^* \mathbf{myLi}(?, \mathbf{myLi}(?, \mathbf{List}(a), a), a) \leq^* \mathbf{myLi}(?, \mathbf{List}(a), a) \leq^* \mathbf{List}(a)$$

We close with a type unification example in TEL. The corresponding example in Java 5.0 is given in Example 6.

*Example 7.* Let  $\mathbf{List}(a) < \mathbf{myLi}(b, a)$  and the start configuration

$$\{\mathbf{List}(\mathbf{List}(\mathbf{Integer})) < \mathbf{List}(x)\}$$

be given. In the first step the reduce 1 rule is applied:  $\{\mathbf{List}(\mathbf{Integer}) < x\}$ . With the fourth step we get the result:

$$\{ \{ x \mapsto \text{List}(\text{Integer}) \}, \{ x \mapsto \text{myLi}(b, \text{Integer}) \} \}.$$

All other infinite numbers of unifiers are instances of these general unifiers (e.g.  $\{ x \mapsto \text{myLi}(\text{List}(\text{Integer}), \text{Integer}) \}$  or  $\{ x \mapsto \text{myLi}(\text{myList}(b, \text{Integer}), \text{Integer}) \}$ ).

## 4 Conclusion and Outlook

In this paper we presented a 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 [3], as the usual unification is the base of type inference in functional programming languages.

With the type unification algorithm, which we have presented in this paper, it would be possible to complete the type inference algorithm of **TEL** [5] respectively to extend the type system of **PROTOS-L** [8] by allowing explicit subtype relationships between polymorphic type constructors.

## References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java<sup>TM</sup> Language Specification. 3rd edn. The Java series. Addison-Wesley (2005)
2. 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
3. Plümicke, M.: Typeless Programming in Java 5.0 with wildcards. In Amaral, V., Veiga, L., Marcelino, L., Cunningham, H.C., eds.: 5th International Conference on Principles and Practices of Programming in Java. ACM International Conference Proceeding Series (September 2007) 73–82
4. Damas, L., Milner, R.: Principal type-schemes for functional programs. Proc. 9th Symposium on Principles of Programming Languages (1982)
5. Smolka, G.: Logic Programming over Polymorphically Order-Sorted Types. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany (May 1989)
6. Hanus, M.: Parametric order-sorted types in logic programming. Proc. TAPSOFT 1991 **LNCS**(394) (1991) 181–200
7. 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
8. Beierle, C.: Type inferencing for polymorphic order-sorted logic programs. In: International Conference on Logic Programming. (1995) 765–779
9. Plümicke, M.: **OBJ-P** The Polymorphic Extension of **OBJ-3**. PhD thesis, University of Tuebingen, WSI-99-4 (1999)
10. Plümicke, M.: Type unification in **Generic-Java**. In Kohlhasse, M., ed.: Proceedings of 18th International Workshop on Unification (UNIF’04). (July 2004)
11. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems 4 (1982) 258–282