Type Unification in Generic–Java

Martin Plümicke

University of Cooperative Education Stuttgart
Department of Information Technology
Florianstraße 15
D–72160 Horb
tel. +49-7451-521142
fax. +49-7451-521190
m.pluemicke@ba-horb.de

Abstract. In this paper we will present a type unification algorithm for type terms of Generic–Java. First we define a type system for expressions in Generic–Java. We give an ordering on the declared classes which represents the inheritance hierarchy. This ordering is extended to an ordering on the set of type terms. Finally the type unification problem on ordered type terms is solved by the type unification algorithm.

1 Introduction

In version 1.5 of sun's java (G–JAVA), generic types are implemented [1, 2]. This means classes can have parameters. The type system is extended by parameterized types, type variables, and type terms. In this paper we present an ordering $\leq^*$ on type terms, which represents the inheritance relation, and the type unification algorithm, which solves the problem to get a substitution σ for two given type terms θ_1, θ_2, such that $\sigma(\theta_1) \leq^* \sigma(\theta_2)$. We call this the *type unification problem*. The solution of the type unification problem is the base of a type inference system for G–JAVA.
First let us consider the types in G–JAVA more detailed. Examples for parameterized types are

```
Seq<A>
Vector<A>
Pair<A,B>
```

The parameters in the classes can be instantiated by each type. For example a declaration of a vector of `Integer`s is done by

```
Vector<Integer> v = new Vector<Integer>();
```

In comparison to the declaration `Vector v = new Vector();` in lower java versions now the vector can only contain instances of the class `Integer`. In most cases datatypes like `Vector` are used only for instances of one class. Nevertheless, if the vector is needed for instances of different classes the vector instances can be declared as `Vector<Object>`.

The main advantage of declarations like this is that many type casts are unnecessary. For example, if we want to get the sixth object of an `Integer` vector `v`, in traditional java (Version ≤ 1.4) we must write

```
Integer i = (Integer)v.get(6);
```

whereas in G–JAVA satisfies

```
Integer i = v.get(6);
```

In parameterized classes the parameters substitute the previous type `Object`, as shown in the following cutout of the class `Vector`:

```
class Vector<A> {

    void add (A elem) { ...}

    A get (int i)  { ...}
}
```

This means that the declaration `Vector<Integer> v = new Vector<Integer>();` instantiated the type variable `A` by `Integer`. Therefore the type cast shown above is unnecessary.
Furthermore some run time errors become compiling errors. In traditional java statements like

```
Vector v = new Vector();
v.add(new Integer(9));
((String)v.get(5)) + "hallo"
```

are type correct, as the result type of `get` is `Object` and the type cast from `Object` to `String` is correct. It leads to an runtime error. In G–JAVA the compilation of the analogous lines

```
Vector<Integer> v = new Vector<Integer>();
v.add(new Integer(9));
((String)v.get(5)) ++ "hallo"
```

causes a type error, as a type cast from `Integer`, the instantiated result type of `get`, to `String` is incorrect.
Now, we give an example for a class with instantiated parameters.

```
class Matrix extends Vector<Vector<Integer>> {

   Matrix mul(Matrix m) {
      Matrix ret = new Matrix ();
      for(int i = 0; i < size(); i++) {
          Vector<Integer> v1 = this.elementAt(i);
          Vector<Integer> v2 = new Vector<Integer> ();
```

```
        for (int j = 0; j < v1.size(); j++) {
            int erg = 0;
            for (int k = 0; k < v1.size(); k++) {
                erg = erg + v1.elementAt(k).intValue()
                    * (m.elementAt(k)).elementAt(j).intValue();
            }
            v2.addElement(new Integer(erg));
        }
        ret.addElement(v2);
    }
    return ret;
}
```

The class `Matrix` is an extension of `Vector<Vector<Integer>>`, where the method `mul` is added.
In the following section we show, how the inheritance hierarchy can be described by polymorphical order-sorted types [3, 4]. In section three we present the type unification algorithm and give some examples. The paper closes with a summary and an outlook.

2 Polymorphic Order–sorted Types as Inheritance Hierarchy

In this section, we define a set of types, an ordering on this set of types, and the translation from the inheritance hierarchy of G–JAVA classes to this type term ordering.
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 Θ of class names and a set of type variables TV. Because of this we will speak of *type terms* instead of types.
Furthermore, $\mathsf{TVar}(\theta)$ denotes the set of type variables occurring in the type term θ.

Definition 1. (**Type terms of declared G–JAVA classes**) *Let JC be a set of declared G–JAVA classes. The rank alphabet $\Theta = (\Theta^{(n)})_{n \in \mathbb{N}}$ is given as: For each class declaration*

$$\texttt{class C<}a_1, \ldots, a_n\texttt{>}\ldots \in JC$$

holds $\texttt{C} \in \Theta^{(n)}$. *The* set of type terms of the declared G–JAVA classes *JC is defined as $T_\Theta(TV)$, the set of terms over the rank alphabet Θ.*
The class names are called type constructors *as the type terms are constructed over the class names.*

In this framework we do not consider F–bounded class parameters of G–JAVA.

Definition 2. (**Type term ordering of declared G–JAVA classes**) *Let JC be a set of declared G–JAVA classes and $T_\Theta(TV)$ the set of type terms. For each declared G–JAVA class in JC*

$$\texttt{class C<}\mathtt{a_1, \ldots, a_n}\texttt{> extends } \tau$$

it holds

$$\texttt{C<}\mathtt{a_1, \ldots, a_n}\texttt{>} \leq \tau.$$

The relation $\leq$ *is called* type term ordering.

The specification of G–JAVA leads to the following properties of a type term ordering.

Lemma 1. *A type term ordering of declared G–JAVA classes has the following properties:*

1. $\leq$ *is finite*
2. *If* $\theta \leq \theta'$ *then* $\mathsf{TVar}(\theta) = \mathsf{TVar}(\theta')$
3. *If* $\theta \leq \theta'$ *then neither* θ *nor* θ' *is a type variable.*

Definition 3. (**Closure** of the type term ordering) *Let* $T_\Theta(TV)$ *be a set of type terms and* $\leq$ *a type term ordering on* $T_\Theta(TV)$. *The closure of the type term ordering* $\leq$ *(denoted by* $\leq^*$ *) is given as the smallest ordering with the conditions:*

– *if* $(\theta, \theta') \in T_\Theta(TV) \times T_\Theta(TV)$ *is an element of the reflexive and transitive closure of* $\leq$ *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, σ_2, *which satisfy the following condition:* $\sigma_1(a) = \sigma_2(a)$ *for all* $a \in \mathsf{TVar}(\theta_2)$.

It is surprising that the condition in the former definition is not $\sigma_1(a) \leq^* \sigma_2(a)$. This is about the contravariance problem in object oriented languages. The following example illustrates this problem.

Example 1. Let the following G–JAVA class be given.

```
class Super { ...}

class Sub extends Super { ...}

class Vector<a> {

    void add (a elem) { ...}
}

class Main {
  public static void main(String[] args) {
    Vector<Super> v;
    v = new Vector<Sub> ();
    v.add(new Super());
  }
}
```

An element of the type `Vector<Sub>` is assigned to the variable `a` of the type `Vector<Super>`. This is no problem as all elements which has the type `Sub` has also the type `Super`. Then a new element of the type `Super` is added to the vector. Now we get the problem as the elements of this vector has the type `Sub` and `Super` is no subtype of `Sub`.

Lemma 2. *The relation* $\leq^*$ *is a partial ordering.*

The partial ordered set $(T_\Theta(TV), \leq^*)$ describes the inheritance hierarchy of a set of declared G–JAVA–classes. In the theory of polymorphic order-sorted algebras [3, 4] such type terms with a partial ordering are denoted as polymorphic order-sorted types.
Later on in the type unification algorithm (section 3.2) we need a restricted closure of the type term ordering, which is finite.

Definition 4. (**Finite Closure** of $\leq$) *Let* $\leq$ *be a type term ordering and* $\leq^*$ *its closure. Then, the* finite closure $\mathbf{FC}(\leq)$ *of* $\leq$ *is defined as the transitive and reflexive closure of*

$$\{\, \overline{\theta} \leq^* \theta' \mid \theta \leq \theta' \,\} \cup \{\, \theta \leq^* \theta'' \mid \theta \leq \theta' \,\}$$

Lemma 3. *The finite closure* $\mathbf{FC}(\leq)$ *is finite.*

Now we give some examples to illustrate the abstract definitions.

Example 2. The well-known inheritance hierarchy of traditional java is given by a type term ordering which is declared on a rank alphabet $(\Theta^{(n)})_{n \in N}$ where $\Theta^{(i)} = \emptyset$ for $i > 0$.

```
class Object { ... }

class Boolean extends Object { ... }

class Character extends Object { ... }

class Number extends Object { ... }

class Integer extends Number { ... }

class String extends Object { ... }
```

From this cutout of the traditional inheritance hierarchy the type term ordering, which is given in fig. 1, is determined.

The second example shows how parameterized classes are translated to the type term ordering.

Example 3. Let the following G–JAVA declarations be given:

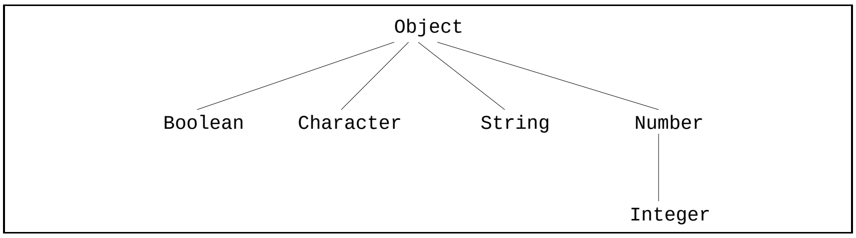

Fig. 1. Inheritance hierarchy of traditional java

```
class List<a> { ...}

class AbstractList<a> extends List<a> { ...}

class Vector<a> extends AbstractList<a> { ...}

class Stack<a> extends Vector<a> { ...}
```

The defined type term ordering $\leq$ is shown in fig. 2. This means that in the

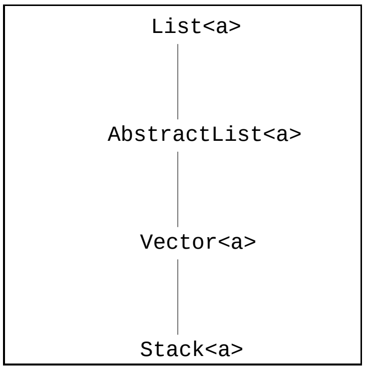

Fig. 2. Type term ordering of a parameterized vector

closure $\leq^*$ of $\leq$ comprises all instances of this type term ordering. For example

$$\texttt{Stack<Integer>} \leq^* \texttt{Vector<Integer>},$$

$$\texttt{Stack<Vector<Integer>>} \leq^* \texttt{Vector<Vector<Integer>>},$$

and

$$\texttt{Vector<List<Integer>>} \leq^* \texttt{List<List<Integer>>}$$

are in the closure of the type term ordering.

The last example shows a finite closure of a type term ordering.

Example 4. Now we change the G–JAVA declaration of the last example:

```
class AbstractList<a> extends List<a> { ...}

class Vector<a> extends AbstractList<a> { ...}

class Matrix<a> extends Vector<Vector<a>> { ...}

class ExtMatrix<a> extends Matrix<a> { ...}
```

We consider now the interesting part of the finite closure $\mathbf{FC}(\leq)$:

$$\begin{aligned} fc_1 = \{ & \texttt{Vector<Vector<a>>} \leq^* \texttt{Vector<Vector<a>>}, \\ & \texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>}, \\ & \texttt{ExtMatrix<a>} \leq^* \texttt{Vector<Vector<a>>} \} \\ fc_2 = \{ & \texttt{Matrix<a>} \leq^* \texttt{Matrix<a>}, \\ & \texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>}, \\ & \texttt{Matrix<a>} \leq^* \texttt{AbstractList<Vector<a>>}, \\ & \texttt{Matrix<a>} \leq^* \texttt{List<Vector<a>>} \} \end{aligned}$$

Then, the finite closure $\mathbf{FC}(\leq) =$ **transitive and reflexive closure of** $(fc_1 \cup fc_2 \cup \leq)$. The interesting part is shown in fig. 3.

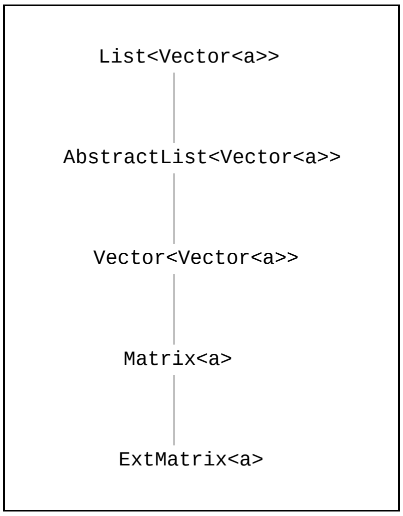

Fig. 3. Part of the finite closure of a type term ordering

Lemma 4. *For each pair* $\overline{\theta} \leq^* \overline{\theta}'$ *there is a pair* $\theta \leq^* \theta' \in \mathbf{FC}(\leq)$ *and a substitution* σ *with* $\sigma(\theta) = \overline{\theta}$ *and* $\sigma(\theta') = \overline{\theta}'$*.*

3 Type Term Unification

Now, we will present the type unification algorithm. First we give an overview of unification algorithms, which are bases of our type unification algorithm. Then we give our algorithm.

3.1 Overview

Polymorphic order-sorted unification is derived on the one hand from the non-polymorphic order-sorted unification [5–7] and on the other hand from the usual unification theory [8–10].
For the polymorphic order-sorted unification a type unification algorithm is needed instead of a simple infimum function as in the non-polymorphic case. The type unification algorithm is developed from the algorithm of Herbrand [8] and A. Martelli and U. Montanari [11, 10].
Unification algorithms for order-sorted terms are well-known from logic programming languages based on a theory of order-sorted algebras. The non-polymorphic order-sorted unification problem is considered in [12, 5–7]. A special case of this problem is solved by Beierle in [13, 14] for the logic programming language PROTOS-L. In [15] Smolka mentions the type unification problem without restrictions as an open problem.
In this paper we solve the type unification problem with the restriction, that from $\theta_1 \leq^* \theta_2$ follows $\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)$ if and only if $\sigma_1(a) = \sigma_2(a)$ for all $a \in \mathsf{TVar}(\theta_2)$. The not restricted type unification problem allows $\sigma_1(a) \leq^* \sigma_2(a)$.

3.2 G–JAVA Type Term Unification

We give an algorithm which determines a substitution σ for two type terms θ_1, θ_2 with

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

We call this the *type unification problem*. The base of our algorithm is the unification algorithm of Herbrand [8] and A. Martelli and U. Montanari [11, 10].
In the following, $T_\Theta(TV)$ denotes a set of type terms, $\leq$ denotes a type term ordering (definition 2), and $\leq^*$ its closure on $T_\Theta(TV)$ (definition 3).

Definition 5. (**Type unifier**) Given type terms $\theta_1, \theta_2 \in T_\Theta(TV)$ and a substitution σ with $\sigma(\theta_1) \leq^* \sigma(\theta_2)$, σ is called a *type unifier*.

Definition 6. (**General type unifier**) Given type terms $\theta_1, \theta_2 \in T_\Theta(TV)$ and a type unifier σ. σ is called *general type unifier* if there is no type unifier σ' and no substitution σ'' such that $\sigma = \sigma' \circ \sigma''$.

Now we present an algorithm, which computes the set of general type unifiers. In this algorithm we unify a set of *equations* $Eq = \{ \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \}$ where $\theta_i, \theta'_j \in T_\Theta(TV)$, and $\theta \lessdot \theta'$ means, that the two type terms should be unified, such that $\sigma(\theta) \leq^* \sigma(\theta')$.
During the unification algorithm $\lessdot$ is replaced by $\doteq$, where $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.

Definition 7. (**Solved form**) A set of equations Eq is in *solved form*, if

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

where $a_1, \ldots, a_n \in TV$ are pairwise different variables, $\theta_1, \ldots, \theta_n \in T_\Theta(TV)$, and for all $i, j \in \{ 1, \ldots, n \}$ $a_i \notin \mathsf{TVar}(\theta_j)$ holds.

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

where

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

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

(erase) $\dfrac{Eq \cup \{ \theta \doteq \theta' \}}{Eq}\ \theta = \theta'$

(swap) $\dfrac{Eq \cup \{ \theta \doteq a \}}{Eq \cup \{ a \doteq \theta \}}\ \theta \notin TV,\ a \in TV$

(adapt) $\dfrac{Eq \cup \{ D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \}}{Eq \cup \{ \theta_1 \doteq \lambda'_1, \ldots, \theta_n \doteq \lambda'_n \}}$

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

- $(D\texttt{<}a_1, \ldots, a_n\texttt{>} \leq^* D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>}) \in \mathbf{FC}(\leq)$ and it holds
- $D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} = D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>}[a_i \mapsto \lambda'_i \mid 1 \leqslant i \leqslant n]$

Fig. 4. G–JAVA type unification

The type unification algorithm The type unification algorithm $\mathbf{TUnify}_{\leq^*}$ is given as follows

Input: Set of equations $Eq = \{ \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \}$
Output: Set of general type unifiers $Uni = \{ \sigma_1, \ldots, \sigma_m \}$

The algorithm itself is given in seven steps:

1. $Eq_1 = \{ \theta \lessdot \theta' \mid (\theta \lessdot \theta') \in Eq \wedge ((\theta, \theta' \notin TV) \vee (\theta, \theta' \in TV)) \}$
2. $Eq_2 = Eq \backslash Eq_1$
3. $Eq_{set} = \{ Eq_1 \} \times (\bigotimes_{(a \lessdot \theta') \in Eq_2} \{ a \doteq \theta \mid (\theta \leq^* \sigma(\theta')) \in \mathbf{FC}(\leq) \})$
$\times (\bigotimes_{(\theta \lessdot a) \in Eq_2} \{ a \doteq \theta' \mid \theta \leq^* \theta' \})$

4. Repeated application of the rules *reduce1*, *reduce2*, *erase*, *swap*, *adapt* of figure 4 to all elements of Eq_{set}. The end configuration Eq'_{set} is reached if for each element no rule is applicable.
5. Application of the following *subst* rule

$$\frac{Eq \cup \{ a \doteq \theta \}}{Eq[a \mapsto \theta] \cup \{ a \doteq \theta \}} \; a \text{ occurs in } Eq \text{ but not in } \theta$$

   to each element of $Eq' \in Eq'_{set}$ as often as possible. The result is Eq''_{set}.
6. (a) Foreach $Eq'' \in Eq''_{set}$ which was changed in the last step starts again with the first step.
   (b) Build the union Eq'''_{set} of all results of (a) and $Eq'' \in Eq''_{set}$ which was not changed in the last step.
7. $Uni = \{ Eq''' \mid Eq''' \in Eq'''_{set} \text{ is in solved form} \}$

The result is then a set of unifiers

$$\{ \sigma_1, \ldots, \sigma_n \},$$

where for each $Eq_i \in Uni$ holds $\sigma_i = \{ a \mapsto \theta \mid a \doteq \theta \}$.
In the following we explain the changes in comparison to the base algorithm. The main difference is that our type unification problem is not unitary, which means that there are more than one general unifier, which solve the problem. This follows from pairs $a \leq^* \theta$ respectively $\theta \leq^* a$, where a is a type variable. The existence of these pairs means that a can be substituted by all type terms which are smaller or equal respectively greater or equal than θ. Each of these types generates an own unifier. These generations are done in steps one, two, and three. After that the known rules of the original unification algorithm are applied. These rules are changed in a few positions:

reduce1 rule: The *reduce1* rule follows from the construction of $\leq^*$ where from

$$C(a_1, \ldots, a_n) \leq D(a_{\pi(1)}, \ldots, a_{\pi(n)})$$

follows $C(\theta_1, \ldots, \theta_n) \leq^* D(\theta'_1, \ldots, \theta'_n)$ only if $\theta_{\pi(i)} = \theta'_i$ for $1 \leqslant i \leqslant n$. This is the reason, why the condition $\lessdot$ of the whole type term is changed to the condition $\doteq$ of the subterms.

reduce2 rule: The *reduce2* rule follows directly from the original algorithm. This is applied to the subterms after one application of the *reduce1* rule.

erase rule: The *erase* rule is unchanged.

swap rule: The *swap* rule is unchanged.

adapt rule: The algorithm which is given if we disclaim of the *adapt* rule solves the type unification problem for type term orderings $\leq$ which satisfy the following condition: If $\lambda_1 \leq \lambda_2$ then the type term λ_1 is of the form $C_1(a_1, \ldots, a_n)$ and the type term λ_2 of the form $C_2(a_{\pi(1)}, \ldots, a_{\pi(n)})$ for classes $C_1, C_2 \in \Theta^{(n)}$ and a set of type variables $\{ a_1, \ldots, a_n \} \subseteq TV$. $\pi : \{ 1, \ldots, n \} \to \{ 1, \ldots, n \}$ is a permutation. This would mean that declarations like

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

are not possible. The rule *adapt* unifies type terms which are built by class declarations like this.

subst rule: The *subst* rule is taken out of step four from the original algorithm. The reason for this is that there are pairs of type terms with type variables $a \leq^* b$. If a or b is substituted, step one respectively step two and three must be applied, which is done by step six (a).

Step six (b) summarizes the different sets of type term pairs. Step seven finally filters these sets of type term pairs which are really unifiers. This means, that they are in the solved form.

Now, we give two examples for the algorithm. The first one shows how more than one unifier is generated.

Example 5. We extend the type term ordering of example 3:

$$\Theta^{(0)} = \{\, \texttt{Integer}, \texttt{Object} \,\}, \Theta^{(1)} = \{\, \texttt{Vector}, \texttt{Stack}, \texttt{AbstractList}, \texttt{List} \,\},$$
$$\Theta^{(2)} = \{\, \texttt{Pair} \,\},$$

$$\texttt{Integer} \leq \texttt{Object},$$

and

$$\texttt{Stack<a>} \leq \texttt{Vector<a>} \leq \texttt{AbstractList<a>} \leq \texttt{List<a>}.$$

Now, we apply the unification algorithm to

$$\begin{array}{l} Eq = \\ \{\, \texttt{a} \lessdot \texttt{Vector<Integer>}, \\ \quad \texttt{AbstractList<Object>} \lessdot \texttt{b}, \\ \quad \texttt{AbstractList<Pair<Stack<Integer>,d>>} \\ \qquad\qquad \lessdot \texttt{List<Pair<c,List<Integer>>>}, \\ \quad \texttt{c} \lessdot \texttt{d} \,\} \end{array}$$

In the first two steps the pairs are filtered which have one type variable and one type term which is no type variable. The result of the first step is

$$\begin{array}{l} Eq_1 = \\ \{\, \texttt{AbstractList<Pair<Stack<Integer>,d>>} \\ \qquad\qquad \lessdot \texttt{List<Pair<c,List<Integer>>>}, \\ \quad \texttt{c} \lessdot \texttt{d} \,\} \end{array}$$

The result of the second step is

$$\begin{array}{l} Eq_2 = \{\, \texttt{a} \lessdot \texttt{Vector<Integer>}, \\ \qquad\quad \texttt{AbstractList<Object>} \lessdot \texttt{b} \} \end{array}$$

In the third step all types which can be mapped to these type variables are determined and the other pairs are added to each of these elements:

$$
\begin{array}{l}
Eq_{set} = \\
\quad \{\, \{\, \texttt{AbstractList<Pair<Stack<Integer>,d>>} \\
\qquad\qquad\qquad \lessdot\ \texttt{List<Pair<c,List<Integer>>>}, \\
\qquad \texttt{c} \lessdot \texttt{d} \,\} \\
\qquad \cup \{\, \texttt{b} \doteq \texttt{AbstractList<Object>},\ \texttt{a} \doteq \texttt{Vector<Integer>} \,\}, \\
\quad\ \{\, \texttt{AbstractList<Pair<Stack<Integer>,d>>} \\
\qquad\qquad\qquad \lessdot\ \texttt{List<Pair<c,List<Integer>>>}, \\
\qquad \texttt{c} \lessdot \texttt{d} \,\} \\
\qquad \cup \{\, \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \doteq \texttt{Vector<Integer>} \,\}, \\
\quad\ \{\, \texttt{AbstractList<Pair<Stack<Integer>,d>>} \\
\qquad\qquad\qquad \lessdot\ \texttt{List<Pair<c,List<Integer>>>}, \\
\qquad \texttt{c} \lessdot \texttt{d} \,\} \\
\qquad \cup \{\, \texttt{b} \doteq \texttt{AbstractList<Object>},\ \texttt{a} \doteq \texttt{Stack<Integer>} \,\}, \\
\quad\ \{\, \texttt{AbstractList<Pair<Stack<Integer>,d>>} \\
\qquad\qquad\qquad \lessdot\ \texttt{List<Pair<c,List<Integer>>>}, \\
\qquad \texttt{c} \lessdot \texttt{d} \,\} \\
\qquad \cup \{\, \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \doteq \texttt{Stack<Integer>} \,\} \,\}
\end{array}
$$

In the fourth step the rule of fig. 4 are applied to the elements of Eq_{set}. The result is

$$
\begin{array}{ll}
Eq'_{set} = \{ & \{\, \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \texttt{c} \lessdot \texttt{d}, \\
& \ \ \texttt{b} \doteq \texttt{AbstractList<Object>},\ \texttt{a} \doteq \texttt{Vector<Integer>} \,\}, \\
& \{\, \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \texttt{c} \lessdot \texttt{d}, \\
& \ \ \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \doteq \texttt{Vector<Integer>} \,\}, \\
& \{\, \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \texttt{c} \lessdot \texttt{d}, \\
& \ \ \texttt{b} \doteq \texttt{AbstractList<Object>},\ \texttt{a} \doteq \texttt{Stack<Integer>} \,\}, \\
& \{\, \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \texttt{c} \lessdot \texttt{d}, \\
& \ \ \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \doteq \texttt{Stack<Integer>} \,\} \,\}
\end{array}
$$

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

$$
\begin{array}{ll}
Eq''_{set} = \{ & \{\, \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \\
& \ \ \texttt{Stack<Integer>} \lessdot \texttt{List<Integer>}, \\
& \ \ \texttt{b} \doteq \texttt{AbstractList<Object>},\ \texttt{a} \doteq \texttt{Vector<Integer>} \,\}, \\
& \{\, \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \\
& \ \ \texttt{Stack<Integer>} \lessdot \texttt{List<Integer>}, \\
& \ \ \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \doteq \texttt{Vector<Integer>} \,\}, \\
& \{\, \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \\
& \ \ \texttt{Stack<Integer>} \lessdot \texttt{List<Integer>}, \\
& \ \ \texttt{b} \doteq \texttt{AbstractList<Object>},\ \texttt{a} \doteq \texttt{Stack<Integer>} \,\}, \\
& \{\, \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \\
& \ \ \texttt{Stack<Integer>} \lessdot \texttt{List<Integer>}, \\
& \ \ \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \doteq \texttt{Stack<Integer>} \,\} \,\}
\end{array}
$$

The sixth step is divided in **(a)** and **(b)**:

(a): In the first three steps of the recursive applying to elements of Eq''_{set} nothing is changed, as Eq_2 is empty.
In step four $\texttt{Stack<Integer>} \lessdot \texttt{List<Integer>}$ is reduced and finally erased.
(b): This leads to the result

$$\begin{aligned} Eq'''_{set} = \{ & \{ \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \\ & \ \texttt{b} \doteq \texttt{AbstractList<Object>},\ \texttt{a} \doteq \texttt{Vector<Integer>} \}, \\ & \{ \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \\ & \ \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \doteq \texttt{Vector<Integer>} \}, \\ & \{ \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \\ & \ \texttt{b} \doteq \texttt{AbstractList<Object>},\ \texttt{a} \doteq \texttt{Stack<Integer>} \}, \\ & \{ \texttt{c} \doteq \texttt{Stack<Integer>}, \texttt{d} \doteq \texttt{List<Integer>}, \\ & \ \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \doteq \texttt{Stack<Integer>} \} \} \end{aligned}$$

In the seventh step nothing is changed, such that the result are four unifiers:

$$\begin{aligned} \{ & \{ \texttt{a} \mapsto \texttt{Vector<Integer>}, \texttt{b} \mapsto \texttt{AbstractList<Object>}, \\ & \ \texttt{c} \mapsto \texttt{Stack<Integer>}, \texttt{d} \mapsto \texttt{List<Integer>} \}, \\ & \{ \texttt{a} \mapsto \texttt{Vector<Integer>}, \texttt{b} \mapsto \texttt{List<Object>}, \\ & \ \texttt{c} \mapsto \texttt{Stack<Integer>}, \texttt{d} \mapsto \texttt{List<Integer>} \}, \\ & \{ \texttt{a} \mapsto \texttt{Stack<Integer>}, \texttt{b} \mapsto \texttt{AbstractList<Object>}, \\ & \ \texttt{c} \mapsto \texttt{Stack<Integer>}, \texttt{d} \mapsto \texttt{List<Integer>} \}, \\ & \{ \texttt{a} \mapsto \texttt{Stack<Integer>}, \texttt{b} \mapsto \texttt{List<Object>}, \\ & \ \texttt{c} \mapsto \texttt{Stack<Integer>}, \texttt{d} \mapsto \texttt{List<Integer>} \} \} \end{aligned}$$

The example do not use the *adapt* rule. We give another example, which shows, how the *adapt* rule works.

Example 6. Now, we change the type term ordering of the above example:

$$\Theta^{(0)} = \{ \texttt{Object} \}, \Theta^{(1)} = \{ \texttt{Vector}, \texttt{Matrix}, \texttt{List} \}$$

and

$$\texttt{Matrix<a>} \leq \texttt{Vector<Vector<a>>}.$$

The relation $\texttt{Matrix<a>} \leq \texttt{Vector<Vector<a>>}$ is declared by a G–JAVA program

```
class Matrix<a> extends Vector<Vector<a>> { ... }
```

Now, we apply the unification algorithm to

$$Eq = \{ \texttt{Matrix<List<b>>} \lessdot \texttt{Vector<Vector<List<Object>>>} \}$$

In the first three steps nothing happens. If we consider in the fourth step the rules of figure 4, we see that neither the *erase*, nore the *swap* rule is applicable. The *reduce1* rule is also not applicable, as there is no relation $\texttt{Matrix<a>} \leq \texttt{Vector<a>}$. Now the *adapt* rule is necessary.
As

$$(\texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>}) \in \mathbf{FC}(\leq)$$

and

$$\texttt{Vector<Vector<List<Object>>>} = \texttt{Vector<Vector<a>>}[\texttt{a} \mapsto \texttt{List<Object>}]$$

by the *adapt* rule follows:

$$Eq_{set} = \{\, \{\, \texttt{List<b>} \doteq \texttt{List<Object>} \,\}\, \}$$

Then the *reduce2* rule leads finally to the result:

$$\{\, \{\, \texttt{b} \mapsto \texttt{Object} \,\}\, \}$$

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

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

Now we give a sketch of the proof.
First we have to prove the *soundness* and then the *completeness*.

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

Completeness For the completeness we have to prove that if there is a unifier of a set of type term pairs then this unifier is also determined by the algorithm. We prove this, by showing for each transformation of the algorithm inductively, that if a substitution is the unifier of a set of type terms before the transformation then the substitution is also a unifier of at least one set of type term pairs after the transformation. As the last transformations transform the set of pairs to a set of pairs in solved form, the completeness ist proved.

4 Conclusion and Future Work

In this paper we presented an unification algorithm, which solves the type unification problem of type terms in G–JAVA (java 1.5). The type term ordering in G–JAVA which is given by the inheritance relation, is continued on the set of type terms. The type unification problem is not unitary.
The type unification problem is base of a type inference system for G–JAVA. As the type unification is not unitary, the type inference algorithm has possibly more than one result for a method. In other words, such a method is overloaded. The future work would be to give an type inference algorithm for G–JAVA. Then, the algorithm would be implemented in a G–JAVA compiler.

References

1. Bracha, G., Cohen, N., Kemper, C., Marx, S., Odersky, M., Panitz, S.E., Thorup, D.S.K., Wadler, P.: Adding Generice to the Java Programming Language: Participant Draft Specification. `http://java.sun.com` (2001)
2. Austin, C.: J2SE 1.5 in a nutshell. `java.sun.com/developer/technicalArticles/releases/j2se15` (2004)
3. Smolka, G.: Logic Programming over Polymorphically Order-Sorted Types. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany (1989)
4. Plümicke, M.: 3.6. In: OBJ–P The Polymorphic Extension of OBJ–3. Volume WSI-99-4. PhD thesis, University of Tuebingen (1999)
5. Meseguer, J., Goguen, J.A., Smolka, G.: Order-sorted unification. Journal of Symbolic Computation **8** (1989) 383–413
6. Smolka, G., Nutt, W., Goguen, J.A., Meseguer, J.: Order-sorted equational computation. In Aït-Kaci, H., Nivat, M., eds.: Resolution of Equations in Algebraic Structures, Volume 2. Academic Press (1989) 297–367
7. Waldmann, U.: Unitary unification in Order-sorted Signatures. Technical Report Forschungsbericht 298, Universität Dortmund (1989 (Revised Version 1990))
8. Herbrand, J.: Recherches sur la Théorie de la Démonstration (thesis). In Goldfarb, W., ed.: Logical Writings, Cambridge (1930)
9. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of ACM **12(1)** (1965) 23–41
10. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems **4** (1982) 258–282
11. Martelli, A., Montanari, U.: Unification in Linear Time and Space, A structured Presentation. Technical Report B76–16, University of Pisa (1976)
12. Schmidt-Schauß, M.: Computational Aspects of an Order-Sorted Logic with Term Declarations. Volume 395 of LNCS. Springer-Verlag (1989)
13. Beierle, C.: Types, Modules and Databases in the Logic Programming Language PROTOS-L. In Bläsius, K., Hendtstück, U., Rollinger, C.R., eds.: Sort and Types in Artificial Intelligence. Volume 418., Springer-Verlag Berlin (1990) 73–110 Proceedings of the workshop, Eringerfeld, April 1989.
14. Beierle, C.: Logic programming with typed unification and its realization on an abstract machine. IBM Journal of Research and Development **36** (1992) 375–390
15. Smolka, G.: Logic programming with polymorphically order-sorted types. Proc. First International Workshop on Algebraic and Logic Programming, Springer-Verlag **LNCS 343** (1988) 53–70 Gaussig, GDR.