# Formalization of the Java 5.0 Type System

Martin Plümicke

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

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

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

is for example a correct type in Java 5.0.
In this paper we present a formalization of this type system. We define the set of correct Java 5.0 type terms, formally. We give a formal definition of the Java 5.0 subtyping ordering. Finally, we consider the properties of the subtyping ordering, which follow from the introduction of wildcards.

## 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 [9]. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principle types.
In [1] the Java 5.0 type system is specified. This specification is done in a semi-formal way. Some definitions are rather formal, as the subtyping relation (§4.10) or the capture conversion (§5.1.10). Other definitions are only given informal, as wildcard types. The presentation is sometimes less clearly arranged.
In this paper we present an integrated framework for the Java 5.0 type system. Without loss of generality we restrict the type system to parameterized reference

types with and without wildcards. We do not consider base types (`int`, `boolean`, `float`, ... ) and raw types.
The paper is organized as follows. In the second section we give the definition of correct Java 5.0 type terms. In the third section we define the subtyping relation, as an extension of the *extends* relation given by the class declarations. In the fourth section we consider the soundness property of the Java 5.0 type system. Finally, we close with a summary and an outlook.

## 2 Java 5.0 Simple Types

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

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

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

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

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

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

As the type terms are constructed over the class names, we call the class names in this framework *type constructors*.
If we consider the Java 5.0 program of Example 1 more accurately, we recognize that the bounds of the type parameters `b` in the class `C` and the bounds of the type parameter `a` in the class `D` are not considered. This leads to the problem that type terms like `C<C<a,b>,a>` are in the term set $T_\Theta(\,TV\,)$, although they are not correct in Java 5.0.
The solution of the problem is the extension of the rank alphabet $\Theta$ to a type signature, where the arity of the type constructors is indexed by bounded type variables. This leads to a restriction in the type term construction, such that the correct set of type terms is a subset of $T_\Theta(\,TV\,)$. Additionally the set of correct type terms is added by some wildcard constructions. We call the set of correct types *set of simple types* $\mathsf{SType}_{TS}(\,BTV\,)$ (Def. 4).
Unfortunately, the definitions of the type signature (Def. 2), the simple types (Def. 4), and the subtyping ordering (Def. 5) are mutually dependent. This is caused by the fact, that the restriction of the set of simple types is defined

by bounded type parameters, whose bounds are also simple types. This means that, for some definitions, we must assume a given set of simple types, without knowing, how the set of simple types is exactly defined.

**Definition 1 (Bounded type variables).** *Let* $\mathsf{SType}_{TS}(\,BTV\,)$ *be a set of simple types. Then, the set of* bounded type variables *is an indexed set* $BTV = (BTV^{(ty)})_{ty \in \mathsf{I}(\,\mathsf{SType}_{TS}(\,BTV\,)\,)}$*, where each type variable is assigned to an intersection of simple types.*
$\mathsf{I}(\,\mathsf{SType}_{TS}(\,BTV\,)\,)$ *denotes the set of intersections over simple types (cp. Def. 4). In the following we will write a type variable* `a` *bounded by the type* $ty$ *as* $\mathtt{a}|_{ty}$*. Type variables which are not bounded can be considered as bounded type variables by* `Object`*.*

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

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

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

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

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

For the following definitions, we need the concept of *capture conversion* ([1] §5.1.10).
In order to define the capture conversion, we have to introduce the *implicit type variables* with lower and upper bounds first. Implicit type variables are used in Java 5.0 during the *capture conversion*, where the wildcards are replaced by implicit type variables. Implicit type variables cannot be used explictly in Java 5.0 programs.
We denote an implicit type variable $T$ with a lower bound $ty$ by ${}_{ty}|T$ and with an upper bound $ty'$ by $T|^{ty'}$.
The *capture conversion* transforms types with wildcard type arguments to equivalent types, where the wildcards are replaced by implicit type variables.

**Definition 3 (Capture conversion).** *Let* $TS = (\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *be a type signature. Furthermore, let be* $C \in TC^{(a_1|_{u_1}, \ldots, a_n|_{u_n})}$ *and* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \in \mathsf{SType}_{TS}(\,BTV\,)$*. Thus, the* capture conversion $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>}$ *of* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ *is defined as:*

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

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

*Example 4.* Let the indexed set of type constructors $TC$ from Example 3 be given again. Then the following holds

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

The following definition of the set of simple types is connected to the corresponding definition of parameterized types in [1], §4.5.

**Definition 4 (Simple types).** *The set of* simple types $\mathsf{SType}_{TS}(\, BTV\,)$ *for a given type signature* $(\mathsf{SType}_{TS}(\, BTV\,),\ TC)$ *is defined as the smallest set satisfying the following conditions:*

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

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

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

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

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

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

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

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

*Example 5.* Let the Java 5.0 program from Example 1 and the corresponding indexed set of type constructors $TC$ from Example 3 be given again. Let additional `Integer` $\in TC^{()}$.
The terms `A<Integer>` and `A<I<Integer>>` are simple types.
From `Integer` $\in TC^{()}$ follows `Integer` is a simple type. As `A` $\in TC^{(\mathtt{a}|_{\mathtt{Object}})}$ with $ty_1 =$ `Integer` follows, that `A<Integer>` is a simple type. From this follows as `I` $\in TC^{(\mathtt{a}|_{\mathtt{Object}})}$ with $ty_1 =$ `I<Integer>`, that `A<I<Integer>>` is also a simple type. As `A<a>` $\leq^*$ `I<a>` the type term `C<A<Integer>, Integer>` is also a simple type. In contrast `C<Integer, Integer>` is no simple type, as `Integer` $\not\leq^*$ `I<Integer>`.

After the definitions of the subtyping relation, we give another example, where the arguments of the type constructors are bounded and wildcards are used.

The set of bounded type variables $BTV$ is in the following extended by the lower and upper bounded type variables.

## 3 Subtyping in Java 5.0

The Java 5.0 inheritance hierarchy consists of two different relations: The "extends relation" (in sign $<$) is explicitly defined in Java 5.0 programs by the *extends*, and the *implements* declarations, respectively. The "subtyping relation" (cp. [1], §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 5 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(BTV)$).** *Let $TS = (\mathsf{SType}_{TS}(BTV), TC)$ be a type signature of a given Java 5.0 program and $<$ the corresponding extends relation. The* subtyping relation $\leq^*$ *is given as the reflexive and transitive closure of the smallest relation satisfying the following conditions:*

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

  *(cp. [1] §4.5.1.1 type argument containment)*
- *Let $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>}$ be the capture conversions of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ and $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ then holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.*

– *For an intersection type* $ty = \theta_1 \,\&\, \ldots \,\&\, \theta_n$ *holds* $ty \leq^* \theta_i$ *for any* $1 \leqslant i \leqslant n$.
– $T|^{(\theta_1 \& \ldots \& \theta_n)} \leq^* \theta_i$ *for any* $1 \leqslant i \leqslant n$.
– $\theta \leq^* {}_\theta|T$

**Corollary 1.** *The subtyping relation is an ordering.*

The following examples illustrates the subtyping definition.

*Example 6.* Let the Java 5.0 program from Example 1 be given again. Then the following relationships hold:

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

The following example shows, how the capture conversions is used.

*Example 7.* Let the subtyping relationship `Vector<Vector<a>>` $\leq^*$ `Matrix<a>` be given. The the following holds:

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

From this follows the question if it holds

$$\texttt{Matrix<}?\texttt{Integer>} \overset{!}{\leq^*} \texttt{Vector<Vector<}?\texttt{Integer>>}$$

or if it holds

$$\texttt{Matrix<}?\texttt{Integer>} \overset{!}{\leq^*} \texttt{Vector<}?\texttt{Vector<}?\texttt{Integer>>}$$

The two Hasse-diagrams presented in Fig. 1 shows that only the second approach is correct. For the supertype construction of `Matrix<`$?$`Integer>` follows by the definition of the subtyping relation that the capture conversion `Matrix<X>`$|^{\texttt{Integer}}$ must be built. This means that, `Vector<Vector<X>`$|^{\texttt{Integer}}$`>>` is a supertype of `Matrix<`$?$`Integer>`. As `Vector<Vector<`$?$`Integer>>` is no supertype of `Vector<Vector<X>`$|^{\texttt{Integer}}$`>>`, it is also no supertype of `Matrix<`$?$`Integer>`. In contrast the simple type `Vector<`$?$`Vector<`$?$`Integer>>` is a supertype of `Vector<Vector<X>`$|^{\texttt{Integer}}$`>>`, which means that `Vector<`$?$`Vector<`$?$`Integer>>` is also supertype of `Matrix<`$?$`Integer>`.

## 4 Soundness of the Java 5.0 type system

Let us consider again the definition of the subtyping relation (Def. 5). 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 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).
Let the following Java 5.0 classes be given.

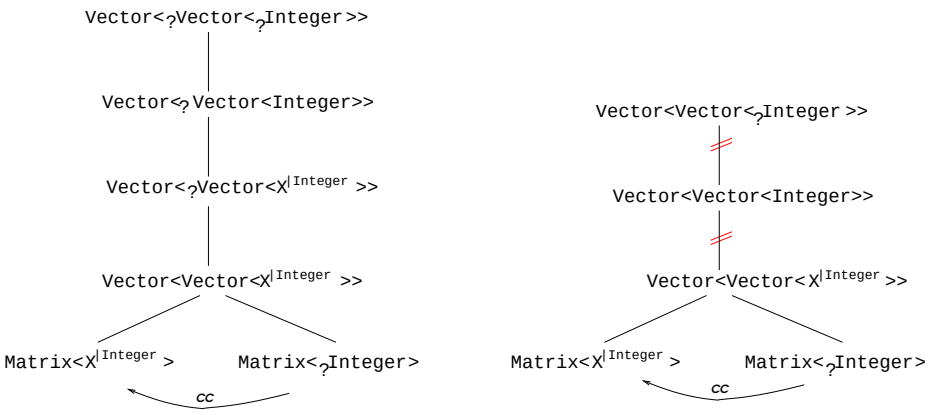


**Fig. 1.** Subtyping relation with capture conversion

```
class Super { ... }
class Sub extends Super { ... }

class Application {
  public static void main(String[] args) {
    Vector<Super> v = new Vector<Sub> ();  //not really correct
    v.addElement(new Super()); }
}
```

An element of the type `Vector<Sub>` is assigned to the variable `v` of the type `Vector<Super>`. This is no problem, as all elements which have the type `Sub` have also the type `Super`. Then a new element of the type `Super` is added to the vector which is assigned to the variable `v`. Now we have the problem, that elements of this vector have the type `Sub` and `Super` is no subtype of `Sub`. If this would be type correct, the type system would be unsound.
As in expression assignments, like `Vector<Super> v = new Vector<Sub>();` the type of the right hand side must be a subtype of the left hand side's type, the subtyping restriction of Def. 5 is introduced. The restriction demands that the declaration must be `Vector<Super> v = new Vector<Super>> ();`. But, sometimes assignments like

```
Vector<Super> v = new Vector<Sub> ();
```

would although be desirable. Therefore wildcards are introduced.
For example it is allowed:

```
Vector<? extends Super> v = new Vector<Sub> ();
```

Now `Vector<Sub>` is a subtype of `Vector<? extends Super>`, which means the assignment is type correct. In this case `v.addElement(new Super())`; is prohibited as `Super` is no subtype of "`? extends Super`". This means that the unsoundness problem is also solved.

On the other hand, if an element of a subclass should be added to a vector of its superclass, the parameter of the vector must have a lower bound:

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

In this case only vectors with a parameter of a supertype of `Super` can be assigned to `v2`. This means that no unsoundness arises.
We have used wildcard types like "`? extends Super`", although there are no simple types. Therefore we have to extend the set of simple type.

**Definition 6 (Extended simple types).** *Let* $\mathsf{SType}_{TS}(\,BTV\,)$ *be a set of simple types. The corresponding set of* extended simple types *is given as*

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

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 (cp. example 8).
Additionally, we have to extend the subtyping relation to wildcard types.

**Definition 7 (Subtyping relation $\leq^*$ on $\mathsf{ExtSType}_{TS}(\,BTV\,)$).** *Let* $\leq^*$ *be a subtyping relation on a given set of simple types* $\mathsf{SType}_{TS}(\,BTV\,)$*. Then* $\leq^*$ *is continued on the corresponding set of extended simple types* $\mathsf{ExtSType}_{TS}(\,BTV\,)$ *by: For* $\theta \leq^* \theta'$*:*

- $\theta \leq^{*\,?} \theta'$,
- ${}_?\theta \leq^* \theta'$, *and*
- ${}_?\theta \leq^{*\,?} \theta'$.

In the following we give two examples, which shows some properties of wildcard types in Java 5.0.

*Example 8.* Let us consider again the class `Vector` with its methods `addElement` and `elementAt` and the classes `Sub` and `Super`.
Let the following declaration be given:

```
Vector<? extends Super> v = new Vector<Sub> ();
```

As said before the methodcall `v.addElement(new Super());` is not correct as $\texttt{Super} \not\leq^* {}_?\texttt{Super}$. But as it holds ${}_?\texttt{Super} \leq^* \texttt{Super}$ the assignment

```
Super sup = v.elementAt(0);
```

is correct.
Vice versa for

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

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

```
Super sup = v2.elementAt(0);   //not really correct
```
is not correct, as $^{?}\texttt{Super} \not\leq^{*} \texttt{Super}$.
Furthermore, the methodcall
```
v2.addElement(v.elementAt(0));
```
is correct, as it holds $_{?}\texttt{Super} \leq^{*} {}^{?}\texttt{Super}$.

*Example 9.* Let the following Java 5.0 program be given:

```
class B<a> { ... }
class C<a> extends B<a> { ... }
class Matrix<a> extends Vector<Vector<a>> { ... }
class ExtMatrix<a> extends Matrix<a> { ... }
class Super { ... }
class Sub extends Super { ... }
```

Now we will give some applications, which show properties of the subtyping ordering and explain that the type system is sound.
The first property is obvious: For all $\theta \leq^{*} \theta'$ holds $\texttt{B<}\theta\texttt{>} \leq^{*} \texttt{B<}_{?}\theta'\texttt{>}$.
This leads to the question, if for any class $Y$ holds also $Y\texttt{<B<}\theta\texttt{>>} \leq^{*} Y\texttt{<B<}_{?}\theta'\texttt{>>}$?
This question can be answered considering the fourth condition of definition 5. As $\texttt{B<}\theta\texttt{>} \neq \texttt{B<}_{?}\theta'\texttt{>}$, the argument type of $Y$ would have to be a wildcard argument. But $\texttt{B<}_{?}\theta'\texttt{>}$ is no wildcard argument. This means that $Y\texttt{<B<}\theta\texttt{>>} \not\leq^{*} Y\texttt{<B<}_{?}\theta'\texttt{>>}$.
If this would be correct, the following Java 5.0 fragment would also be correct

```
Vector<B<? extends Super>> v = new Vector<B<Sub>> (); //is not really
                                                     //correct
v.addElement(new B<Super>());
```

If this would be correct, the type system would be unsound, as an element of the type `B<Super>` is assigned to a vector of elements of the type `B<Sub>`.
But for any $Y$ holds obviously $Y\texttt{<B<}\theta\texttt{>>} \leq^{*} Y\texttt{<}_{?}B\texttt{<}_{?}\theta'\texttt{>>}$.
The next question is, if it holds $\texttt{Matrix<}\theta\texttt{>} \leq^{*} \texttt{Vector<Vector<}_{?}\theta'\texttt{>>}$ for $\theta \leq^{*} \theta'$?
As $\texttt{Matrix<a>} \leq^{*} \texttt{Vector<Vector<a>>}$ for a type variable $\texttt{a} \in BTV$ holds $\texttt{Matrix<}\theta\texttt{>} \leq^{*} \texttt{Vector<Vector<}\theta\texttt{>>}$. But

$$\texttt{Matrix<}\theta\texttt{>} \leq^{*} \texttt{Vector<Vector<}\theta\texttt{>>} \not\leq^{*} \texttt{Vector<Vector<}_{?}\theta'\texttt{>>}$$

(cp. Example 7). This means that $\texttt{Matrix<}\theta\texttt{>} \not\leq^{*} \texttt{Vector<Vector<}_{?}\theta'\texttt{>>}$.
We will also consider, what would happen if it would be correct:

```
Vector<Vector<? extends Super>> v = new Matrix<Sub>(); //is not really
                                                       //correct
v.addElement(new Vector<Super>());
```

In this case the type system would also be not sound, as an element of the type `Vector<Super>` is assigned to a matrix of elements of the type `Sub`.
A further question arises if we consider again that for $\theta \in \mathsf{SType}_{TS}(\,BTV\,)$ holds $\texttt{Matrix<}\theta\texttt{>} \leq^{*} \texttt{Vector<Vector<}\theta\texttt{>>}$ . The question is, if it holds also $\texttt{Matrix<}_{?}\theta\texttt{>} \leq^{*} \texttt{Vector<Vector<}_{?}\theta\texttt{>>}$? As $_{?}\theta \notin \mathsf{SType}_{TS}(\,BTV\,)$ for $a \in BTV$ from $\texttt{Matrix<a>} \leq^{*} \texttt{Vector<Vector<a>>}$ does not follow $\texttt{Matrix<}_{?}\theta\texttt{>} \leq^{*} \texttt{Vector<Vector<}_{?}\theta\texttt{>>}$.
For this we consider again an application:

```
Vector<Vector<? extends Super>> v;
Matrix<? extends Super> w = new Matrix<Sub>();
v = w; //is not really correct
v.addElement(new Vector<Super>());
```

This application shows again that the type system would not be sound. Therefore $\texttt{Matrix<}_?\theta\texttt{>} \not\leq^* \texttt{Vector<Vector<}_?\theta\texttt{>>}$ and the assignment `v = w` is not type correct.
But $\texttt{Matrix<}_?\theta\texttt{>} \leq^* \texttt{Vector<}_?\texttt{Vector<}_?\theta\texttt{>>}$ holds. As the capture conversions of $\texttt{Matrix<}_?\theta\texttt{>}$ is $\texttt{Matrix<T}|^\theta\texttt{>}$ and $T|^\theta \in \mathsf{SType}_{TS}(\,BTV\,)$ follows, that $\texttt{Matrix<T}|^\theta\texttt{>} \leq^* \texttt{Vector<Vector<T}|^\theta\texttt{>>}$. With $\texttt{Vector<Vector<T}|^\theta\texttt{>>} \leq^* \texttt{Vector<}_?\texttt{Vector<T}|^\theta\texttt{>>} \leq^* \texttt{Vector<}_?\texttt{Vector<}_?\texttt{T}|^\theta\texttt{>>} \leq^* \texttt{Vector<}_?\texttt{Vector<}_?\theta\texttt{>>}$ follows $\texttt{Matrix<}_?\theta\texttt{>} \leq^* \texttt{Vector<}_?\texttt{Vector<}_?\theta\texttt{>>}$.

Often the properties covariance respectively contravariance of type constructors are considered in object-oriented languages. Java 5.0 type constructors are neither covariant nor contravariant. The following corollary shows corresponding properties in Java 5.0.

**Corollary 2 (Subtyping properties).** *For two simple types $\theta \leq^* \theta'$ and a type constructor* `Cl` *(class name) holds*

- $\texttt{Cl<}\theta\texttt{>} \not\leq^* \texttt{Cl<}\theta'\texttt{>}$
- $\texttt{Cl<}\theta'\texttt{>} \not\leq^* \texttt{Cl<}\theta\texttt{>}$
- $\texttt{? extends } \theta \leq^* \theta'$, *but* $\texttt{Cl<}\theta\texttt{>} \leq^* \texttt{Cl<? extends } \theta'\texttt{>}$.
- $(\texttt{? extends})\ \theta \leq^* \texttt{? super } \theta'$ *but* $\texttt{CL<(? super) } \theta'\texttt{>} \leq^* \texttt{CL<? super } \theta\texttt{>}$.

## 5 Conclusion and Outlook

In this paper we presented a formalization of the Java 5.0 type system. We defined the set of Java 5.0 simple types as type terms, which are explicitly allowed in Java 5.0 programs. We extended this set by wildcard types, which appear implicitly during the type checking. We defined a subtyping ordering at first on the set of Java 5.0 simple types and extended it to wildcard types. Additionally, we considered the soundness of the Java 5.0 type system. We showed, how the Java 5.0 type system becomes quite flexible by introducing the wildcards without loosing the soundness.
The Java 5.0 type system is the base for the definition of a type inference algorithm. We will give a type inference algorithm for Java 5.0 type terms with wildcards. Furthermore, we will implement this system.

## References


1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java$^{TM}$ Language Specification. 3rd edn. The Java series. Addison-Wesley (2005)
2. Damas, L., Milner, R.: Principal type-schemes for functional programs. Proc. 9th Symposium on Principles of Programming Languages (1982)

3. Smolka, G.: Logic Programming over Polymorphically Order-Sorted Types. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany (May 1989)
4. Hanus, M.: Parametric order-sorted types in logic programming. Proc. TAPSOFT 1991 **LNCS**(394) (1991) 181–200
5. Hill, P.M., Topor, R.W.: A Semantics for Typed Logic Programs. In Pfenning, F., ed.: Types in Logic Programming. MIT Press (1992) 1–62
6. Beierle, C.: Type inferencing for polymorphic order-sorted logic programs. In: International Conference on Logic Programming. (1995) 765–779
7. Plümicke, M.: OBJ–P The Polymorphic Extension of OBJ–3. PhD thesis, University of Tuebingen, WSI-99-4 (1999)
8. Plümicke, M.: Type unification in Generic–Java. In Kohlhase, M., ed.: Proceedings of 18th International Workshop on Unification (UNIF'04). (July 2004)
9. Plümicke, M., Bäuerle, J.: Typeless Programming in Java 5.0. In Gitzel, R., Aleksey, M., Schader, M., Krintz, C., eds.: 4th International Conference on Principles and Practices of Programming in Java. ACM International Conference Proceeding Series, Mannheim University Press (August 2006) 175–181
10. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems **4** (1982) 258–282