Subtyping in Java 5.0

Martin Plümicke
University of Cooperative Education Stuttgart/Horb
m.pluemicke@ba-horb.de

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

Vectors<? extends Vector<AbstractList<Integer>>>

is a correct type in Java 5.0.

Considering 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]. 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. The presentation is sometimes less clearly arranged. In this paper we present a summary of an integrated framework for the Java 5.0 type system.

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 \( TV \), where the index \( n \) represents the number of type parameters. Therefore we denote them as type terms instead of types.

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

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

The rank alphabet \( \Theta = \Theta^{(n)}_{n \in \mathbb{N}} \) is determined by \( \Theta^{(1)} = \{ A, B, I, J \} \) and \( \Theta^{(2)} = \{ C \} \). For example \( A<\text{Integer}>, A<B<\text{Boolean}>>, \) and \( C<A<\text{Object}>, \text{Object} \) 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 bound of the type parameters $b$ in the class $C$ is not considered. This leads to the problem that type terms like $C<\langle a, b \rangle, a\rangle$ 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 $\text{SType}_{TS}(BTV)$ (Def. 3).

**Definition 1 (Bounded type variables).** Let $\text{SType}_{TS}(BTV)$ be a set of simple types. Then, the set of bounded type variables is an indexed set $BTV = \langle BTV^{(\Theta)} \rangle_{\Theta \in \text{SType}_{TS}(BTV)}$, where each type variable is assigned to an intersection of simple types $\cap \text{SType}_{TS}(BTV)$ denotes the set of intersections over simple types. In the following we will write a type variable $a$ bounded by the type $ty$ as $a_{ty}$.

Type variables which are not bounded can be considered as bounded type variables by $\text{Object}$.

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

```java
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 $BTV^{(\text{Number})} = \{ a, r \}$ and $BTV^{(\text{Vector<Integer> & J<a> & I})} = \{ t \}$.

**Definition 2 (Type signature, type constructor).** Let $\text{SType}_{TS}(BTV)$ be a set of simple types. A type signature $TS$ is a pair $(\text{SType}_{TS}(BTV), TC)$ 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 $TC^{(a,b)} = \{ A, B, I, J \}$, $TC^{(a,b)} = \{ C \}$, and $TC^{(a,b,c)} = \{ D \}$.

For the following definitions, we need the concept of capture conversion ([1] §5.1.10). The capture conversion of $C<\theta_1, \ldots, \theta_n>$ is denoted by $CC(C<\theta_1, \ldots, \theta_n>)$.

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

**Definition 3 (Simple types).** The set of simple types $\text{SType}_{TS}(BTV)$ for a given type signature $(\text{SType}_{TS}(BTV), TC)$ is defined as the smallest set satisfying the following conditions:
35

- For each intersection type ty: $BTy^{(a)} \subseteq STy_{TS}(BTV)$
- $TC^0 \subseteq STy_{TS}(BTV)$
- For $ty_i \in STy_{TS}(BTV)$ \cup
  \{?\} \cup \{? \text{ extends } \tau \mid \tau \in STy_{TS}(BTV)\}
  \cup \{? \text{ super } \tau \mid \tau \in STy_{TS}(BTV)\}$
  and $C \in TC^0_{\alpha_1 \ldots \alpha_n}$ holds
  $C<ty_1, \ldots, ty_n> \in STy_{TS}(BTV)$ if after $C<ty_1, \ldots, ty_n>$ subjected to
  the capture conversion resulting in the type $C<\bar{ty}_1, \ldots, \bar{ty}_n>$, for each actual
  type argument $\bar{ty}_i$ holds: $\bar{ty}_i \leq^* b_i[a_j \mapsto \bar{ty}_j \mid 1 \leq j \leq n]$, where $\leq^*$ is a
  subtyping ordering (Def. 4).

Example 4. Let the Java 5.0 program from Example 1 and the corresponding
indexed set of type constructors $TC$ from Example 3 be given again. Let additional
Integer, Boolean $\in TC^0$.
The terms $\text{A<Integer> are A<B<Boolean>> are simple types.}$ $C<A<Object>, Object>$
is also is a simple type as $A<Object> \leq^* I<Object>$.
In contrast, $C<A, b>, a>$ is no simple type, as $C<a, b> \not\leq^* I<a>$.

3 Subtyping in Java 5.0

In the following we derive the “subtyping relation” (cp. [1], §4.10) as the reflexive,
transitive, and instatiating closure of the extends/implements relation given by
the Java 5.0 program.
We will use $\text{extends}$ as an abbreviation for the type term “? extends $\theta$” and $\text{super}$ as an
abbreviation for the type term “? super $\theta$”.

Definition 4 (Subtyping relation $\leq^*$ on $STy_{TS}(BTV)$). Let $TS =$
$(STy_{TS}(BTV), TC)$ be a type signature of a given Java 5.0 program. The
subtyping relation $\leq^*$ is given as the reflexive and transitive closure of the
smallest relation satisfying the following conditions:

- if $\theta$ extends/implements $\theta'$ then $\theta \leq^* \theta'$.
- if $\theta_1 \leq \theta_2$ then $\sigma_1(\theta_1) \leq \sigma_2(\theta_2)$ for all substitutions $\sigma_1, \sigma_2 : BTV \rightarrow$
  $STy_{TS}(BTV)$, where for each type variable $a$ of $\theta_2$ holds $\sigma_1(a) = \sigma_2(a)$
  (soundness condition).
- $\alpha \leq^* \theta_i$ for $a \in BTV^{(\theta_1 \& \ldots \& \theta_n)}$ and $1 \leq i \leq n$
- It holds $C<\theta_1, \ldots, \theta_n> \leq^* C<\theta'_1, \ldots, \theta'_n>$ if for each $\theta_i$ and $\theta'_i$, respectively,
  one of the following conditions is valid:
    - $\theta_i = \gamma_{\theta_i}, \theta'_i = \gamma_{\theta'_i}$ and $\theta_i \leq^* \theta'_i$.
    - $\theta_i = \theta_i[\bar{\theta}_j], \theta'_i = \bar{\theta}_j$ and $\bar{\theta}_j \leq^* \theta_i$.
    - $\theta_i, \theta'_i \in STy_{TS}(BTV)$ and $\theta_i = \theta'_i$.
    - $\theta'_i = \gamma_{\theta_i}$
    - $\theta'_i = \theta_i$

\footnote{For non wildcard type arguments the capture conversion $\bar{ty}_i$ equals $ty_i$.}
(cp. [1] §4.5.1.1 type argument containment)
- Let $C<\theta_1, \ldots, \theta_n>$ be the capture conversions of $C<\theta_1, \ldots, \theta_n>$ and $C<\theta'_1, \ldots, \theta'_n>$
  $\leq^* C<\theta'_1, \ldots, \theta'_n>$ then holds $C<\theta_1, \ldots, \theta_n> \leq^* C<\theta'_1, \ldots, \theta'_n>$.
- For an intersection type $ty = \theta_1 \& \ldots \& \theta_n$ holds $ty \leq^* \theta_i$ for any $1 \leq i \leq n$.

Corollary 1. The subtyping relation is an ordering.

**Example 5.** 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 $A\langle a \rangle$ implements $I\langle a \rangle$
- $A\langle Integer \rangle \leq^* I\langle Integer \rangle$, where $\sigma_1 = [a \mapsto Integer] = \sigma_2$
- $A\langle Integer \rangle \leq^* I?\langle Object \rangle$, as Integer $\leq^* Object$
- $A\langle Object \rangle \leq^* I?\langle super Integer \rangle$, as Integer $\leq^* Object$

4 Soundness of the Java 5.0 type system

Let us consider again the definition of the subtyping relation (Def. 4). It is surprising that the soundness 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.

```java
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.

Therefore expression assignments like this are prohibited. The restriction demands that the declaration must be $v = \text{Vector<Super>} v = \text{new Vector<Super>} ()$; But, sometimes assignments like $v = \text{new Vector<Sub>()}$; would although be preferable. Therefore wildcards are introduced. For example the declaration $\text{Vector<}? \text{ extends Super} > v = \text{new Vector<Sub> ()}$; is allowed. Now $\text{Vector<Sub>}$ is a subtype of $\text{Vector<}? \text{ extends Super} >$, which means the assignment is type correct. In this case $v\text{.addElement(new Super());}$ 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 5 (Extended simple types).** Let STypeTS(BTV) be a set of simple types. The corresponding set of extended simple types is given as

\[
\text{ExtSType}_{TS}(BTV) = \text{SType}_{TS}(BTV) \\
\cup \{ ? \} \cup \{ ? \text{ extends } \theta \mid \theta \in \text{SType}_{TS}(BTV) \} \\
\cup \{ ? \text{ super } \theta \mid \theta \in \text{SType}_{TS}(BTV) \}.
\]

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.

Finally, we have to extend the subtyping relation to wildcard types.

**Definition 6 (Subtyping relation \(\leq^*\) on ExtSType_{TS}(BTV)).** Let \(\leq^*\) be a subtyping relation on a given set of simple types SType_{TS}(BTV). Then \(\leq^*\) is continued on the corresponding set of extended simple types ExtSType_{TS}(BTV) by: For \(\theta \leq^* \theta'\) holds: \(\theta \leq^* ? \theta', \text{ } \gamma \theta \leq^* \theta'\), and \(\gamma \theta \leq^* ? \theta'.\)

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 wildcards without loosing the soundness.

The Java 5.0 type system is the base of the type inference algorithm [2]. We have implemented a prototype of this type inference algorithm.

References