Subtyping in Java 5.0

Martin Plümicke

University of Cooperative Education
Stuttgart/Horb

20. Februar 2008

Overview

Type terms in Java 5.0:

explicitly used:

```
Vector<Vector<Integer>>
Vector<? extends List<Object>>
Vector<? super List<Object>>
```

only inferred:

```
? extends List<Object>
? super List<Object>
```

Type terms

Example:

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

Type terms

Example:

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

- `A<Integer>`, `A<B<Boolean>>`, `C<A<Object>,Object>` $\in T_\Theta(\ TV\ )$

Type terms

Example:

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

- A<Integer>, A<B<Boolean>>, C<A<Object>, Object> $\in T_\Theta(\ TV\ )$
- C<C<a>, a> $\in T_\Theta(\ TV\ )$, **but no Java 5.0 type term!!!**

Bounded type variables

Bounded type variables

Definition:

- $BTV = (BTV^{(ty)})_{ty}$ (set of bounded type variables)
  Notation: $a|_{ty}$ means a is bounded by the (intersection) type ty.

Bounded type variables

Definition:

- $BTV = (BTV^{(ty)})_{ty}$ (set of bounded type variables)
  Notation: $a|_{ty}$ means a is bounded by the (intersection) type ty.

Example:

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

- $BTV^{(\texttt{Number})} = \{\, \texttt{a}, \texttt{r} \,\}$
- $BTV^{(\texttt{Vector<Integer>\,\&\,J<a>\,\&\,I})} = \{\, \texttt{t} \,\}$

Type signature

Type signature

Definition *Type signature*:

- $TS = (TC^{(tv_1 \ldots tv_n)})_{tv_i \in BTV}$ (BTV^*–indexed set of *type constructors*)

Type signature

Definition *Type signature*:

- $TS = (TC^{(tv_1 \ldots tv_n)})_{tv_i \in BTV}$ (BTV^*–indexed set of *type constructors*)

Example:

```
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> { ... }
```

- $TC^{(\texttt{a}|_{\texttt{Object}})} = \{\texttt{A}, \texttt{B}, \texttt{I}, \texttt{J}\}$
- $TC^{(\texttt{a}|_{\texttt{I<b>}}\ \texttt{b}|_{\texttt{Object}})} = \{\texttt{C}\}$
- $TC^{(\texttt{a}|_{\texttt{B<a>\&J<b>}}\ \texttt{b}|_{\texttt{Object}})} = \{\texttt{D}\}$

Simple types $\mathsf{SType}_{TS}(\,BTV\,)$ (first approach)

- $BTV \subseteq \mathsf{SType}_{TS}(\,BTV\,)$
- $TC^{()} \subseteq \mathsf{SType}_{TS}(\,BTV\,)$
- For $C \in TC^{(a_1|_{ty_1} \ldots a_n|_{ty_n})}$ and a substitution σ

$$C\!<\!\sigma(\,a_1\,),\ldots,\sigma(\,a_n\,)\!> \in \mathsf{SType}_{TS}(\,BTV\,),$$

if $\sigma(\,a_i\,) \leq^* \sigma(\,ty_i\,)$ ($\leq^*$ is the subtyping relation).

Example:

```
class A<a> implements I<a> { ... }

class B<a> extends A<a> { ... }

class C<a extends I<b>,b> { ... }

interface I<a> { ... }
```

- $TC^{(\texttt{a}|_{\texttt{Object}})} = \{\texttt{A}, \texttt{B}, \texttt{I}\}$
  $TC^{(\texttt{a}|_{\texttt{I<b>}}\ \texttt{b}|_{\texttt{Object}})} = \{\texttt{C}\}$
- A<Integer>, A<B<Boolean>>, C<A<Object>, Object> $\in \mathsf{SType}_{TS}(\mathit{BTV})$
- C<C<a>, a> $\notin \mathsf{SType}_{TS}(\mathit{BTV})$

Subtyping ordering $\leq^*$

Definition:

- if θ `extends`/`implements` θ' 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 with $\sigma_1(a) = \sigma_2(a)$ (soundness condition).

Subtyping ordering $\leq^*$

Definition:

- if θ `extends/implements` θ' 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 with $\sigma_1(a) = \sigma_2(a)$ (soundness condition).

Example:

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

- `A<a>` $\leq^*$ `I<a>`, as `A<a> implements I<a>`
- `A<Integer>` $\leq^*$ `I<Integer>`, where $\sigma_1 = [a \mapsto \texttt{Integer}] = \sigma_2$
- `A<Integer>` $\not\leq^*$ `I<Object>`, although indeed `Integer` $\leq^*$ `Object`, but $\sigma_1(\texttt{a}) \neq \sigma_2(\texttt{a})$.

Soundness condition: $\sigma_1(a) = \sigma_2(a)$

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

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

Soundness condition: $\sigma_1(a) = \sigma_2(a)$

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

class Application {
  public static void main(String[] args) {
    Vector<Super> v = new Vector<Sub> ();  //not allowed
    v.addElement(new Super());     //wrong: Super ⊴* Sub
  }

  void meth (Vector<Sub> subvec) {
    Vector<Super> supervec = subvec; //could make sense
    ...                               //but not allowed
  }
}
```

Introduction of wildcards

`extends`–**wildcards**

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

Means: The elements of the vector `v` are subtypes of `Super`.

Introduction of wildcards

`extends`–**wildcards**

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

Means: The elements of the vector `v` are subtypes of `Super`.

$\Rightarrow$

- `Vector<Sub>` $\leq^*$ `Vector<? extends Super>`

Introduction of wildcards

`extends`–**wildcards**

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

Means: The elements of the vector `v` are subtypes of `Super`.

$\Rightarrow$

- `Vector<Sub>` $\leq^*$ `Vector<? extends Super>`
- `Super` $\not\leq^*$ `? extends Super`
- `v.addElement(new Super());` not valid

Introduction of wildcards

`extends`–**wildcards**

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

Means: The elements of the vector `v` are subtypes of `Super`.

$\Rightarrow$

- `Vector<Sub>` $\leq^*$ `Vector<? extends Super>`
- `Super` $\not\leq^*$ `? extends Super`
- `v.addElement(new Super());` not valid
- `? extends Super` $\leq^*$ `Super`
- `Super sup = v.elementAt(0);` valid

super–wildcards

Problem:

```
void addSuperElement(Vector<Sub> v) {
    v.addElement(new Sub ());
}
```

Application of `addSuperElement(new Vector<Super> ());` is not allowed, although it make sense.

super–wildcards

Problem:

```
void addSuperElement(Vector<Sub> v) {
    v.addElement(new Sub ());
}
```

Application of `addSuperElement(new Vector<Super> ());` is not allowed, although it make sense.

Solution:

```
void addSuperElement(Vector<? super Sub> v) {
    v.addElement(new Sub ());
}
```

Now `addSuperElement(new Vector<Super> ());` is allowed.

Properties of `super`–wildcards

- `Vector<Super>` $\leq^*$ `Vector<? super Sub>`, as `Sub` $\leq^*$ `Super`.

Properties of super–wildcards

- `Vector<Super>` $\leq^*$ `Vector<? super Sub>`, as `Sub` $\leq^*$ `Super`.
- `? super Sub` $\not\leq^*$ `Sub`
- `Vector<? super Sub> v;`
  `Sub e = v.elementAt(0);` not valid.

Properties of super–wildcards

- `Vector<Super>` $\leq^*$ `Vector<? super Sub>`, as `Sub` $\leq^*$ `Super`.
- `? super Sub` $\not\leq^*$ `Sub`
- `Vector<? super Sub> v;`
  `Sub e = v.elementAt(0);` not valid.
- `Sub` $\leq^*$ `? super Sub`
- `Vector<? super Sub> v;`
  `v.addElement(new Sub());` valid.

Simple types $\mathsf{SType}_{TS}(\mathit{BTV})$

- $\mathit{BTV} \subseteq \mathsf{SType}_{TS}(\mathit{BTV})$
- $TC^{()} \subseteq \mathsf{SType}_{TS}(\mathit{BTV})$
- For $ty_i \in \mathsf{SType}_{TS}(\mathit{BTV})$
  $\cup \{ \texttt{?} \}$
  $\cup \{ \texttt{? extends } \tau \mid \tau \in \mathsf{SType}_{TS}(\mathit{BTV}) \}$
  $\cup \{ \texttt{? super } \tau \mid \tau \in \mathsf{SType}_{TS}(\mathit{BTV}) \}$
  and $C \in TC^{(a_1|\texttt{Object} \ldots a_n|\texttt{Object})}$ holds

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

Abbreviations for wildcards:

Instead of `A<? extends B>` we write

`A<?B>`

and instead of `C<? super D>` we write

`C<?D>`.

Subtyping ordering $\leq^*$ (extension)

- if θ `extends`/`implements` θ' then $\underline{\theta \leq^* \theta'}$.
- if $\theta_1 \leq^* \theta_2$ then $\underline{\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)}$, where for each type variable a of θ_2 holds $\sigma_1(a) = \sigma_2(a) \in \mathsf{SType}_{TS}(BTV)$ (soundness condition).
- It holds $\underline{C\langle\theta_1, \ldots, \theta_n\rangle \leq^* C\langle\theta'_1, \ldots, \theta'_n\rangle}$ if for θ_i and θ'_i either
  - $\theta_i = {}_?\overline{\theta}_i$, $\theta'_i = {}_?\overline{\theta}'_i$ and $\overline{\theta}_i \leq^* \overline{\theta}'_i$ or
  - $\theta_i = {}^?\overline{\theta}_i$, $\theta'_i = {}^?\overline{\theta}'_i$ and $\overline{\theta}'_i \leq^* \overline{\theta}_i$ or
  - θ_i, θ'_i are no wildcard arguments and $\theta_i = \theta'_i$ or
  - $\theta'_i = {}_?\theta_i$ or
  - $\theta'_i = {}^?\theta_i$

Motivation for a continuation of $\leq^*$ on wildcard types

An element should be read from a vector of a wildcard type:

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

$\Longrightarrow$ `? extends Super` $\leq^*$ `Super`

Motivation for a continuation of $\leq^*$ on wildcard types

An element should be read from a vector of a wildcard type:

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

$\Longrightarrow$ `? extends Super` $\leq^*$ `Super`

An element of a subclass should be added to a vector of a superclass:

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

$\Longrightarrow$ `Super, Sub` $\leq^*$ `? super Super`

Extended simple types

$\mathsf{SType}_{TS}(\mathit{BTV})$ is a set of simple types:
The corresponding set of *extended simple types* is given as

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

Extended simple types

$\mathsf{SType}_{TS}(\mathit{BTV})$ is a set of simple types:
The corresponding set of *extended simple types* is given as

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

Wildcard types cannot be used, explicitly. They are only inferred.

$\leq^*$ on $\mathrm{ExtSType}_{TS}(\mathit{BTV})$

For $\theta, \theta' \in \mathrm{SType}_{TS}(\mathit{BTV})$ with $\theta \leq^* \theta'$ holds:

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

Conclusion

- Formalization of the Java 5.0 type system
- Subtyping ordering on $\mathrm{SType}_{TS}(\mathit{BTV})$ and $\mathrm{ExtSType}_{TS}(\mathit{BTV})$
- Base for the type inference algorithm, which works, if
  - there is a subtyping ordering and
  - there is a type unification algorithm