Type unification for structural types in Java

– Extended Abstract –

Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
pl@dhbw.de

In the past we considered type inference for Java with generics and lambda-expressions. The base of our algorithm was a finitary type unification. The algorithm determines nominal types in subjection to a given environment. This is a hard restriction as separate compilation of Java classes without relying on type informations of other classes is impossible. Let us consider the following example:

```
import java.util.Vector;
class A { m (v) { return v.elementAt(0); } }
```

For the method `m` the type `Vector<A>` $\rightarrow$ `A` is inferred, as `Vector` is the only class in the environment. This type is not principal. The principal type of `m` would be a structural type `ST<A>`, that have a method `elementAt`: `ST<A>` $\rightarrow$ `A`.
We present an extended type unification algorithm as the base of a type inference algorithm for a Java-like language, that infers structural types without given environments.

The type unification algorithm

The *type unification problem* is given as: For a set of constraints $\{\, \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \,\}$, where θ_i, θ'_j are type terms, a substitution σ is demanded, such that for all $1 \leqslant i \leqslant n : \sigma(\,\theta_i\,)$ is a subtype of $\sigma(\,\theta'_i\,)$. The substitution σ is called *type unifier*. The type unification algorithm is given by eight rules, that are applied most often as possible. If the result C is in solved form (all elements has either the form $T \doteq \theta$, $T \lessdot \theta$, or $\theta \lessdot T$, where T is a type variable) then C is the result otherwise the algorithm fails. We prove the termination of the algorithm and give a soundness and a completeness theorem.

Example

Extending the example from the beginning

```
class Vector<A> extends ST<A> {  given in as Standard Java }

class Main { main() { return new A<>().m(new Vector<Integer>(...)); }}
```

leads to the set of type constraints $C = \{\, \texttt{Vector<Integer>} \lessdot \nu_1, \nu_1 \lessdot \texttt{ST<}\nu_2\texttt{>} \,\}$. Applying the type unification algorithm to C results in $\{\, \nu_2 \doteq \texttt{Integer}, \texttt{Vector<Integer>} \lessdot \nu_1, \nu_1 \lessdot \texttt{ST<Integer>} \,\}$. The type inferred program is then given as

```
class A { ν2 m (ν1 v) { return v.elementAt(0); } }

class Main { Integer main() { return new A<>().m(new Vector<Integer>(...)); }}
```