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:

```java
import java.util.Vector;

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

For the method \( m \) the type \( \text{Vector}<A> \rightarrow A \) is inferred, as \( \text{Vector} \) is the only class in the environment. This type is not principal. The principal type of \( m \) would be a structural type \( \text{ST}<A> \), that have a method \( \text{elementAt}: \text{ST}<A> \rightarrow A \).

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 \( \{ \theta_1 \preceq \theta'_1, \ldots, \theta_n \preceq \theta'_n \} \), where \( \theta_i, \theta'_j \) are type terms, a substitution \( \sigma \) is demanded, such that for all \( 1 \leq i \leq n \) : \( \sigma(\theta_i) \) is a subtype of \( \sigma(\theta'_i) \). The substitution \( \sigma \) 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 \( T = \theta, T \preceq \theta \), or \( \theta \preceq T \), 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

```java
import java.util.Vector;

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

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

```java
class A { \nu_2 \text{ m} (\nu_1 \text{ v}) \{ \text{return} \text{v.elementAt(0)}; \} }
```

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