Type unification for structural types in Java

Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract

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. In this paper 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.

1 Introduction

We considered in different contributions in the past type inference in Java-like languages e.g. [Plü15]. The resulting types are nominal types, that are dependent from a given environment. 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 `Vector<A> → 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: SA<A> → A`. We gave a sketch of a type inference algorithm for structural types in [Plü10]. Our algorithm is a generalization and an improvement of an idea, that has given in [ADDZ05]. We replace a complex linking algorithm by type unification. As in [ADDZ05] the algorithm is given for a Featherweight Java. Basically, we reduce the Java type inference for nominal types to type unification, as in [DM82] the type inference for ML was reduced to ordinary unification [Rob65]. While we gave in [Plü04] type unification for G-JAVA (Java with generics but without wildcards), in [Plü09] we presented type unification for Java with wildcards. Both unifications problems are in general not unitary but finitary. This results from the property that the soltutions of a constraint $T \ll ty'$, where $T$ is type variable, $ty'$ is a Java type and $\ll$ means should be a subtype, are all substitutions $\{ T \mapsto ty \mid ty$ is a subtypes of $ty' \}$. The corresponding type unification algorithms are advancements of Martelli and Montanari’s [MM82].

In a similar way we will reduce Java type inference for structural types to a changed type unification, where $T \ll ty'$ (resp. $ty \ll T$) itself is a result, that is not resolved.

In the following we give some basic definitions. Then we present the type unification algorithm and give an example. Finally, we close with a summary.

2 Basic definitions

The parametrized Java classes and interfaces form a rank alphabet $\Theta = (\Theta^{(\cdot)})_{n \in \mathbb{N}}$ of its names, where $n$ represents the number of class-parameters. In the following $TV$ is a set of type variables.
Definition 1 (Set of type terms). Let $\Theta = (\Theta^{(n)})_{n\in\mathbb{N}}$ be the rank alphabet of classes and interfaces. The set of type terms $T_{\Theta}(TV)$ is given as the set of terms over the rank alphabet $\Theta$ and the set of type variables $TV$.

The extends/implements relation induces the type term ordering.

Definition 2 (Type term ordering). Let $T_{\Theta}(TV)$ be a set of type terms and $<$ the extends/implements relation on $T_{\Theta}(TV)$. The Type term ordering $\leq^*$ is given as the smallest ordering with the conditions:

- if $(\theta, \theta') \in T_{\Theta}(TV) \times T_{\Theta}(TV)$ is an element of the reflexive and transitive closure of $<$ then $\theta \leq^* \theta'$.

- if $\theta_1 \leq^* \theta_2$ then $\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)$ for all substitutions $\sigma_1, \sigma_2$, which satisfy the following condition: $\sigma_1(a) = \sigma_2(a)$ for all $a \in TVar(\theta_2)$.

The type unification problem is given in the following definition.

Definition 3 (Type unification problem, type unifier). The type unification problem is given as: For a set of constraints $\{\theta_1 \triangleleft \theta_1', \ldots, \theta_n \triangleleft \theta_n',\}$, where $\theta_i, \theta_i' \in T_{\Theta}(TV)$, a substitution $\sigma$ is demanded, such that for all $1 \leq i \leq n: \sigma(\theta_i) \leq^* \sigma(\theta_i')$. The substitution $\sigma$ is called type unifier.

During the unification algorithm $\triangleleft$ is replaced by $\triangleq$, where $\triangleq \triangleq \triangleleft'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.

Definition 4 (Most general type unifier). A given type unifier $\sigma$ of $C$ is called most general type unifier if for any type unifier $\sigma'$ of $C$ there is a substitution $\sigma''$ such that $\sigma' = \sigma'' \circ \sigma$.

If the type unifications algorithm does not fail the result is given in solved form. The following definition of solved form is an extension of the definition for the ordinary unification.

Definition 5 (Solved form). A set of constraints $C$ is in solved form, if all elements of $C$ has either the form $T \triangleq \theta$, $T \triangleq \theta$, or $\theta \triangleq T$, where $T$ is a type variable, $\theta \in T_{\Theta}(TV)$, and $T \not\in TVar(\theta)$ holds.

3 The type unification algorithm

The algorithm $TUnify(C)$ is given by the rules (Figure 1) application the most often as possible. If $C$ is finally in solved form then $C$ is the result, otherwise the algorithm fails.

Lemma 6 (Termination). The algorithm $TUnify$ terminates.

Lemma 7 (Soundness of $TUnify$). If a substitution $\sigma$ is a solution of a constraint set $C$ then $\sigma$ is also a solution of $TUnify(C)$.

Lemma 8 (Completeness, most general unifier of $TUnify$). Let for a set of constraints $C$ the substitution $\sigma = \{T \mapsto ty | T \triangleq ty \in TUnify(C)\}$ be given. For any solution $\sigma'$ of $C$ there are substitutions $\sigma''$ and $\sigma_{rest}$, such that $\sigma' = \sigma'' \circ (\sigma_{rest} \circ \sigma) \cup \sigma_{rest}$. The substitution $(\sigma_{rest} \circ \sigma) \cup \sigma_{rest}$ is a most general unifier in dependency on $\sigma_{rest}$.

Remark The substitution $\sigma_{rest}$ is a solution of the remaining pairs $T \triangleq ty$ and $ty \triangleq T$.
solve

### 3.1 Type inference algorithm

The **type inference algorithm** is given by the three following functions **TYPE**, **construct**, and **solve**:

- **TYPE** collects the type constraints.
- **construct** builds the interfaces, that represent the structural types.
- **solve** unifies the constraints by the type unification algorithm.

---

Unify is ordinary unification given in [Rob65] or more efficient in [MM82]
4 Example

Let the following typeless Java class be given

```java
class A { mt(x, y, z) { return x.sub(y).add(z); } }
```

The result of the type inference algorithm is

```java
interface Sub<R, T> { R sub(T x); }
interface Add<R, T> { R add(T x); }
class A <ν₁, ν₃, ν₄, ν₆>
    [ν₃ extends ν₅, ν₄ extends ν₇, ν₁ extends Sub<ν₂, ν₅>, ν₂ extends Add<ν₆, ν₇>]
    ν₆ mt(ν₁ x, ν₃ y, ν₄ z) { return x.sub(y).add(z); }
```

In this case the type unification function `solve` changes nothing as the result constraints set of `construct` \( \{ ν₃ ⪯ ν₅, ν₄ ⪯ ν₇, ν₁ ⪯ Sub<ν₂, ν₅>, ν₂ ⪯ Add<ν₆, ν₇> \} \) is already in solved form. These constraints are given as bounds in the output syntax of `A`. The four class-parameters are the argument- and the return-types of the method `mt`. The interfaces `Add` and `Sub` represents the structural types.

Now we give a class `myInteger`, that implements `Sub` as well as `Add`.

```java
class myInteger extends Sub<myInteger, myInteger>, Add<myInteger, myInteger> {
    Integer i;
    myInteger sub(myInteger x) { return new myInteger(i - x.i); }
    myInteger add(myInteger x) { return new myInteger(i + x.i); }
}
```

Finally, in the class `Main` an instance of `A` is used and the method `mt` is called.

```java
class Main {
    main() { return new A<>()
        .mt(new myInteger(2), new myInteger(1), new myInteger(3)); }
}
```

The result of `TYPE` and `construct` is the constraint set, that has to be unified:

\[
C_{\text{main}} = \{ ν₃ ⪯ ν₅, ν₄ ⪯ ν₇, ν₁ ⪯ Sub<ν₂, ν₅>, ν₂ ⪯ Add<ν₆, ν₇>,
    myInteger ⪯ ν₁, myInteger ⪯ ν₃, myInteger ⪯ ν₄ \}
\]

The class declarations implies the subtyping ordering `myInteger ⪯ Sub<myInteger, myInteger>` and `myInteger ⪯ Add<myInteger, myInteger>`.

The type unification algorithm applied to `C_{\text{main}}` is given as follows: With the `adapt2`-rule follows from `myInteger ⪯ ν₁, ν₁ ⪯ Sub<ν₂, ν₅>`: `myInteger ⪯ ν₁, ν₁ ⪯ Sub<myInteger, myInteger>`, \( ν₂ = \text{myInteger}, ν₅ = \text{myInteger} \). From this follows with the `subst`-rule `myInteger ⪯ Add<ν₆, ν₇>` and with the `adapt1`-rule: `Add<myInteger, myInteger> ⪯ Add<ν₆, ν₇>`. With the `reduce`- and `swap`-rule we get: \( ν₆ = \text{myInteger}, ν₇ = \text{myInteger} \). With the `subst`-rule follows `myInteger ⪯ ν₃`, \( ν₃ ⪯ \text{myInteger} \) and `myInteger ⪯ ν₄, ν₄ ⪯ \text{myInteger}` and from this with the `refl`-rule: \( ν₃ = \text{myInteger} \) and `ν₄ = \text{myInteger}`.

The output syntax of the type inference algorithm differs from standard Java. On the one hand the bounds of the class-parameters are given in a additional declaration in `[ ]-bracket. On the other hand type variables, that are only necessary to describe the bounds and not used in the class fields or methods need not to be declared explicitly as class-parameters.
Type unification for structural types in Java  

Martin Plümicke

The result of \texttt{solve} is given as:
\[
\begin{align*}
\{ & \text{myInteger} < \nu_1, \nu_1 < \text{Sub<myInteger, myInteger>} \\
& \nu_2 = \text{myInteger}, \nu_5 = \text{myInteger}, \nu_6 = \text{myInteger}, \\
& \nu_7 = \text{myInteger}, \nu_3 = \text{myInteger}, \nu_4 = \text{myInteger} \}
\end{align*}
\]

The resulting Java class is given as:
\[
\text{class Main [ myInteger extends } \nu_1, \nu_1 \text{ extends Sub<myInteger, myInteger> ]}
\begin{align*}
\text{main}() \{ \\
\text{ return new } A<>().mt(\text{new myInteger(2), new myInteger(1), new myInteger(3)}); \}
\end{align*}
\]

There is one remaining type variable \(\nu_1\), that is not used in a argument- or return-type of a method. Therefore \(\nu_1\) is no class-parameter of \texttt{Main}. The two remaining bounds of \(\nu_1\) are consistent. This means \texttt{main} is executable and the result of execution is 4.

5 Summary

We have presented a type unification algorithm as the base of a type inference algorithm for a Java-like language, that infers structural types. The presented type unification is an extension of the type unification for nominal types. While in the nominal case all solutions of a constraint \(a < \theta\) are determined, in the structural case the constraint \(a < \theta\) itself is a result, that is resolved not until an instance of the corresponding class is created. This approach has two main advantages: It reduces the number of unification solutions enormously and allows separate compilation of Java classes without relying on type information of other classes.

References


