Type unification for structural types in Java

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

June 26, 2016
Overview

Introduction

Motivation

Type inference algorithm

Type unification
  The type unification problem
  The type unification algorithm
  Type unification rules

Example

Summary
  Conclusion
  Future work
The Java-TX System (basic idea)

- Type inference for Java in similar way as in functional programming languages (Haskell, ocaml, sml, ...)

No type annotations necessary, however static typing
The Java-TX System (basic idea)

- **Type inference for Java** in similar way as in functional programming languages (Haskell, ocaml, sml, ...)

  No type annotations necessary, however static typing

- Type inference in functional programming languages is reduced to ordinary unification (Damas/Miler 1982 ⇒ Robinson 1965)
The Java-TX System (basic idea)

- Type inference for Java in similar way as in functional programming languages (Haskell, ocaml, sml, ...)
  
  No type annotations necessary, however static typing

- Type inference in functional programming languages is reduced to ordinary unification (Damas/Miler 1982 ⇒ Robinson 1965)

- Java type inference is reduced to type unification
Java type unification

For two type terms $\theta_1$ and $\theta_2$ a substitution $\sigma$ is demanded such that:

$$\sigma(\theta_1) \leq^* \sigma(\theta_2).$$
Java type unification

For two type terms $\theta_1$ and $\theta_2$ a substitution $\sigma$ is demanded such that:

$$\sigma(\theta_1) \leq^* \sigma(\theta_2).$$

**UNIF’ 04:** Java type terms without wildcards (PIZZA’s type system)
Java type unification

For two type terms $\theta_1$ and $\theta_2$ a substitution $\sigma$ is demanded such that:

$$\sigma(\theta_1) \leq^* \sigma(\theta_2).$$

**UNIF’ 04:** Java type terms without wildcards (PIZZA’s type system)

**UNIF’ 07:** Java type terms with wildcards (Java 5.0’s type system)
Eclipse plugin
class A {
    m (v) {
        return v.elementAt(0);
    }
}
Motivation

Type inference example

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

No type inference result for m
import java.util.Vector;

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

class A {

    m (v) {
        return v.elementAt(0);
    }
}

**Result:** m : Vector<T> → T (Nominal type).
Motivation

Type inference example

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

Could be inferred a structural type?
Motivation

Type inference example

```java
interface I<T> { T elementAt(int x); }

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

Result: m : I<T> → T (Structural type).
```
Motivation

Costs

Type unification is not unitary, but finitary

There is a totally correct algorithm. But the number of solution grows exponentially:

\[ n^m \]

\( n \): Number of classes (types) in the environment

\( m \): Number of the types in the input terms
Type inference algorithm

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

- **TYPE** collects the type constraints.

- **construct** builds the interfaces, that represent the structural types.

- **solve** unifies the constraints by the type unification algorithm.
The type unification problem

Definitions:

\( \theta \leq^* \theta' \): \( \theta \) is a subtype of \( \theta' \).

\( \theta < \theta' \): \( \theta \) and \( \theta' \) should be unified, such that \( \sigma(\theta) \leq^* \sigma(\theta') \).

\( \theta \equiv \theta' \): \( \theta \) and \( \theta' \) should be unified, such that \( \sigma(\theta) = \sigma(\theta') \).
The type unification problem

Definitions:

\( \theta \leq^* \theta' \): \( \theta \) is a subtype of \( \theta' \).

\( \theta < \theta' \): \( \theta \) and \( \theta' \) should be unified, such that \( \sigma(\theta) \leq^* \sigma(\theta') \).

\( \theta \cong \theta' \): \( \theta \) and \( \theta' \) should be unified, such that \( \sigma(\theta) = \sigma(\theta') \).

Type unification problem:

For a given set of type term pairs

\[
\{ \theta_1 < \theta_1', \ldots, \theta_n < \theta_n' \}
\]

a unifier (substitution) \( \sigma \) is demanded, such that

\[
\sigma(\theta_i) \leq^* \sigma(\theta_i')
\]
Why is type unification not unitary?

This is caused by pairs like:

\[ a < \theta' \]

The results are

\[ \{ a \mapsto \theta \mid \theta \leq^* \theta' \} \].
Type Unification with structural types

Pairs like

\[ a < \theta \]

are themselves results, that means

all subtypes of \( \theta' \) (not explicitly given)

are results.
The type unification algorithm

**Base:** An Efficient Unification Algorithm [Martelli, Montanari 1982]
The type unification algorithm

**Base:** An Efficient Unification Algorithm [Martelli, Montanari 1982]

The algorithm \textbf{TUnify}(C) is given by the following type unification rules. Application the most often as possible.

If \(C\) is finally in solved form (all elements has the form \(T/\theta\), \(T/\theta\), or \(\theta/T\), where \(T\) is a type variable and \(\theta\) is a type term) then \(C\) is the result, otherwise the algorithm fails.
The type unification algorithm

**Base:** An Efficient Unification Algorithm [Martelli, Montanari 1982]

The algorithm **TUnify**( C ) is given by the following type unification rules application the most often as possible.

If C is finally in solved form (all elements has the form \( T \equiv \theta \), \( T \prec \theta \), or \( \theta \prec T \), where \( T \) is a type variable and \( \theta \) is a type term) then C is the result, otherwise the algorithm fails.
Type unification rules

**reduce**

\[
C \cup \{ D<\theta_1, \ldots, \theta_n> \ll D'<\theta'_1, \ldots, \theta'_n> \}
\]

\[
C \cup \{ \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \}
\]

where \( D<T_1, \ldots, T_n> \leq^* D'<T_1, \ldots, T_n> \) with \( T_i \) are type variables

**adapt1**

\[
C \cup \{ D<\theta_1, \ldots, \theta_n> \ll D'<\theta'_1, \ldots, \theta'_m> \}
\]

\[
C \cup \{ D'<\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m>[ T_i \mapsto \theta_i \mid 1 \leq i \leq n] \doteq D'<\theta'_1, \ldots, \theta'_m> \}
\]

where \((D<T_1, \ldots, T_n> \leq^* D'<\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m>)\) with \( T_i \) are type variables
adapt2

\[ C \cup \{ D^{<\theta_1, \ldots, \theta_n>} < S_1, S_1 < S_2, \ldots, S_{k-1} < S_k, S_k < D'^{<\theta'_1, \ldots, \theta'_m>}, S_k \} \cup \{ \sigma( D^{<\theta_1, \ldots, \theta_n>}) < S_1, S_1 < S_2, \ldots, S_{k-1} < S_k, S_k < \sigma( D'^{<\theta'_1, \ldots, \theta'_m>}) \} \cup \sigma \]

where

- \( k \geq 1 \)
- \( S_i \in TV \) and
- \( (D^{<\theta_1, \ldots, \theta_n>} \not\leq D'^{<\theta'_1, \ldots, \theta'_m>}) \) but
  \( (D^{<T_1, \ldots, T_n>} \leq D'^{<\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m>}) \) with \( T_i \in TV \)
- \( \sigma = \text{Unify}( \{ D'^{<\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m>}[T_i \mapsto \theta_i \mid 1 \leq i \leq n] \doteq D'^{<\theta'_1, \ldots, \theta'_m>} \} ) \)
erase1 \[ \frac{C \cup \{ \theta \leq \theta' \}}{C} \] \quad \theta \leq^* \theta'

erase2 \[ \frac{C \cup \{ \theta \equiv \theta' \}}{C} \] \quad \theta = \theta'

swap \[ \frac{C \cup \{ \theta \equiv T \}}{C \cup \{ T \equiv \theta \}} \] \quad \theta \notin TV, \ T \in TV

subst \[ \frac{C \cup \{ T \equiv \theta \}}{C[T \mapsto \theta] \cup \{ T \equiv \theta \}} \] \quad T \in TV \text{ and } T \text{ occurs in } C \text{ but not in } \theta

refl \[ \frac{C \cup \{ \theta \leq T_1, T_1 \leq T_2, \ldots, T_{n-1} \leq T_n, T_n \leq \theta \}}{C \cup \{ T_i = \theta \mid 1 \leq i \leq n \}} \] \quad \theta \notin TV, \ T_i \in TV
Example

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

The result of the type inference algorithm

```java
interface Sub<R, T> { R sub(T x); }
interface Add<R, T> { R add(T x); }

class A <\nu_1,\nu_3,\nu_4,\nu_6>
    [\nu_3 \text{ extends } \nu_5,
     \nu_4 \text{ extends } \nu_7,
     \nu_1 \text{ extends } \text{Sub}<\nu_2,\nu_5>,
     \nu_2 \text{ extends } \text{Add}<\nu_6,\nu_7>]
    {
        \nu_6 \text{ mt}(\nu_1 \text{ x}, \nu_3 \text{ y}, \nu_4 \text{ z})
        {
            \text{return } \nu_1 \text{ x}.\text{sub}(\nu_3 \text{ y}).\text{add}(\nu_4 \text{ z});
        }
    }
```
The result of the type inference algorithm

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);
    }}

The application of solve (unification) changes nothing, as
{ ν₃ < ν₅, ν₄ < ν₇, ν₁ < Sub<ν₂,ν₅>, ν₂ < Add<ν₆,ν₇> } is in solved form.
Interface implementation: Sub and Add

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);
    }
}
class A $<\nu_1, \nu_3, \nu_4, \nu_5>$
    $[\nu_3 < \nu_5, \nu_4 < \nu_7, \nu_1 < \text{Sub}<\nu_2, \nu_5>, \nu_2 < \text{Add}<\nu_6, \nu_7>]$\{ 
    \nu_6 \text{ mt}(\nu_1 \ x, \ \nu_3 \ y, \ \nu_4 \ z) \{ 
    \text{return} \ x.\text{sub}(y).\text{add}(z); 
    \}
    \}
}\}

class Main \{

    main() \{ 
    \text{return} \ \text{new} \ A<>() 
    .\text{mt}(\text{new} \ \text{myInteger}(2), 
    \text{new} \ \text{myInteger}(1), 
    \text{new} \ \text{myInteger}(3)); 
    \}
    \}
\}
The result of TYPE and construct

\[
C = \{ ~ \nu_3 < \nu_5, \\
         \nu_4 < \nu_7, \\
         \nu_1 < \text{Sub}<\text{myInteger}, \nu_5>, \\
         \nu_2 < \text{Add}<\nu_6, \nu_7>, \\
         \text{myInteger} < \nu_1, \\
         \text{myInteger} < \nu_3, \\
         \text{myInteger} < \nu_4 \} \\
\]

is to be unified.
Unification

\[ C = \{ \nu_3 < \nu_5, \nu_4 < \nu_7, \nu_1 < \text{Sub}\langle\text{myInteger}, \nu_5\rangle, \nu_2 < \text{Add}\langle\nu_6, \nu_7\rangle, \\
\text{myInteger} < \nu_1, \text{myInteger} < \nu_3, \text{myInteger} < \nu_4 \} \]
Unification

\[ C = \{ \nu_3 < \nu_5, \nu_4 < \nu_7, \nu_1 < \text{Sub<myInteger,} \nu_5>, \nu_2 < \text{Add<} \nu_6, \nu_7>, \text{myInteger} < \nu_1, \text{myInteger} < \nu_3, \text{myInteger} < \nu_4 \} \]

With the *adapt2*-rule follows from \text{myInteger} < \nu_1, \nu_1 < \text{Sub<} \nu_2, \nu_5>: 

\[
\begin{align*}
\text{myInteger} < \nu_1, \nu_1 & < \text{Sub<myInteger, myInteger>}, \\
\nu_2 & \doteq \text{myInteger}, \nu_5 & \doteq \text{myInteger}.
\end{align*}
\]
Unification

\[ C = \{ \nu_3 < \nu_5, \nu_4 < \nu_7, \nu_1 < \text{Sub}<\text{myInteger},\nu_5>, \nu_2 < \text{Add}<\nu_6,\nu_7>, \text{myInteger} < \nu_1, \text{myInteger} < \nu_3, \text{myInteger} < \nu_4 \} \]

With the \textit{adapt2}-rule follows from \text{myInteger} < \nu_1, \nu_1 < \text{Sub}<\nu_2,\nu_5>:

\[
\begin{align*}
\text{myInteger} < \nu_1, & \quad \nu_1 < \text{Sub}<\text{myInteger},\text{myInteger}>, \\
\nu_2 & \triangleq \text{myInteger}, \quad \nu_3 & \triangleq \text{myInteger}.
\end{align*}
\]

From this follows with the \textit{subst}-rule \text{myInteger} < \text{Add}<\nu_6,\nu_7> and with the \textit{adapt1}-rule: \text{Add}<\text{myInteger},\text{myInteger}> \triangleq \text{Add}<\nu_6,\nu_7>.
Unification

\[ C = \{ \nu_3 < \nu_5, \nu_4 < \nu_7, \nu_1 < \text{Sub<myInteger,}\nu_5>, \nu_2 < \text{Add<\nu_6,}\nu_7>, \\
\text{myInteger < } \nu_1, \text{myInteger < } \nu_3, \text{myInteger < } \nu_4 \} \]

With the \textit{adapt2-rule} follows from \text{myInteger < } \nu_1, \nu_1 < \text{Sub<}\nu_2,\nu_5>:

\[ \text{myInteger < } \nu_1, \nu_1 < \text{Sub<myInteger,myInteger>,} \]
\[ \nu_2 \doteq \text{myInteger}, \nu_5 \doteq \text{myInteger}. \]

From this follows with the \textit{subst-rule} \text{myInteger < } \text{Add<\nu_6,\nu_7>} and with the \textit{adapt1-rule}: \text{Add<myInteger,myInteger> \doteq Add<\nu_6,\nu_7>}.

With the \textit{reduce-} and \textit{swap-rule} follows: \nu_6 \doteq \text{myInteger}, \nu_7 \doteq \text{myInteger}
Unification

\[ C = \{ \nu_3 < \nu_5, \nu_4 < \nu_7, \nu_1 < \text{Sub}<\text{myInteger},\nu_5>, \nu_2 < \text{Add}<\nu_6,\nu_7>, \text{myInteger} < \nu_1, \text{myInteger} < \nu_3, \text{myInteger} < \nu_4 \} \]

With the adapt2-rule follows from \text{myInteger} < \nu_1, \nu_1 < \text{Sub}<\nu_2,\nu_5>:

\[
\text{myInteger} < \nu_1, \nu_1 < \text{Sub}<\text{myInteger},\text{myInteger}>, \nu_2 \doteq \text{myInteger}, \nu_5 \doteq \text{myInteger}.
\]

From this follows with the subst-rule \text{myInteger} < \text{Add}<\nu_6,\nu_7> and with the adapt1-rule: \text{Add}<\text{myInteger},\text{myInteger}> \doteq \text{Add}<\nu_6,\nu_7>.

With the reduce- and swap-rule follows: \nu_6 \doteq \text{myInteger}, \nu_7 \doteq \text{myInteger}

With the subst-rule follows

\[
\text{myInteger} < \nu_3, \nu_3 < \text{myInteger}, \text{myInteger} < \nu_4, \nu_4 < \text{myInteger}
\]

and from this with the refl-rule:

\[
\nu_3 \doteq \text{myInteger}, \nu_4 \doteq \text{myInteger}.
\]
The result of solve

The result of solve is given as:

\[
\{ \text{myInteger} \triangleq \nu_1, \nu_1 \triangleq \text{Sub<myInteger,myInteger>}, \\
\nu_2 \triangleq \text{myInteger}, \nu_5 \triangleq \text{myInteger}, \nu_6 \triangleq \text{myInteger}, \\
\nu_7 \triangleq \text{myInteger}, \nu_3 \triangleq \text{myInteger}, \nu_4 \triangleq \text{myInteger} \}
\]
The result of solve

The result of solve is given as:

\[
\{ myInteger < \nu_1, \nu_1 \prec Sub<myInteger, myInteger>, \\
\nu_2 \preceq myInteger, \nu_5 \preceq myInteger, \nu_6 \preceq myInteger, \\
\nu_7 \preceq myInteger, \nu_3 \preceq myInteger, \nu_4 \preceq myInteger \}
\]

class Main [myInteger extends \nu_1, \\
\nu_1 extends Sub<myInteger, myInteger> ] {

    myInteger main() {
        return new A<>()
            .mt(new myInteger(2),
                new myInteger(1),
                new myInteger(3));
    }
}
Soundness and Completeness Theorem

**Soundness**: If a substitution $\sigma$ is a solution of a constraint set $C$ then $\sigma$ is also a solution of $\text{TUnify}(C)$. 
Soundness and Completeness Theorem

**Soundness:** If a substitution $\sigma$ is a solution of a constraint set $C$ then $\sigma$ is also a solution of $\text{TUnify}(C)$.

**Completeness:** Let

- $C$ be a set of constraints,
- $\sigma'$ a solution of $C$ and
- $\sigma = \{ T \mapsto ty \mid T \vdash ty \in \text{TUnify}(C) \}$

Then there are substitutions $\sigma''$ and $\sigma_{rest}$, such that

$$\sigma' = \sigma'' \circ \left( (\sigma_{rest} \circ \sigma) \cup \sigma_{rest} \right)$$

($\sigma_{rest}$ is a solution of the remaining pairs $T \preceq ty$ and $ty \preceq T$)
Related Work

Ancona, Damiani, Drossopoulou, Zucca: *Polymorphic bytecode: Compositional compilation for Java-like languages*, POPL '05

- no type inference for argument and return types
- compilation to polymorphic bytecode with type constraints
- linking with constraint solving
We have presented a type unification for Java structural types.

- Extension of [Martelli and Montanari 1982 and Pluemicke 2007]
- type unification is unitary
- separate compilation of Java classes without relying on type informations of other classes.
- infers structural types, given as generated interfaces
Future work

- Complete prototypical implementation (Eclipse-PlugIn)
- Optimize unification:
  - Nominal types: different solutions
  - Structural types: one solution