Type unification for structural types in Java

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

June 26, 2016

Overview

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 $\Rightarrow$ 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 $\Rightarrow$ Robinson 1965)

- Java type inference is reduced to type unification

Java type unification

For two type terms θ_1 and θ_2 a substitution σ is demanded such that:

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

Java type unification

For two type terms θ_1 and θ_2 a substitution σ 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

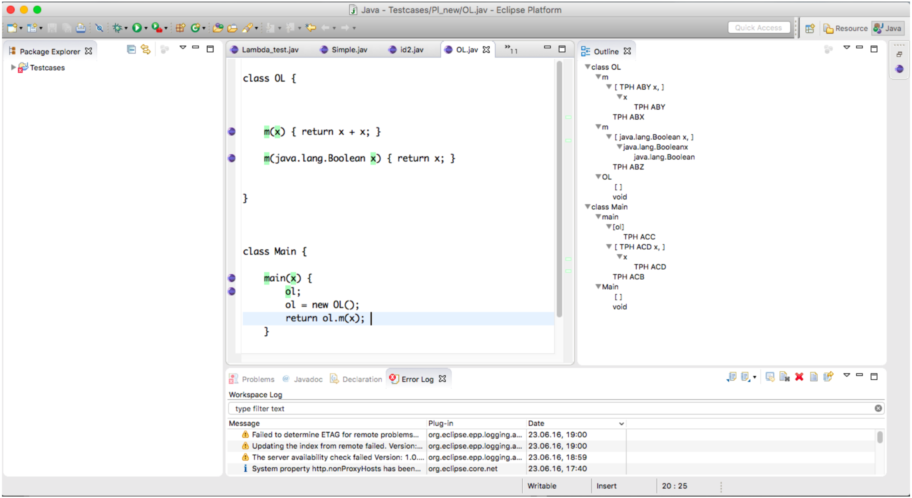
For two type terms θ_1 and θ_2 a substitution σ is demanded such that:

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

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



Motivation

Type inference example

```
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`

Motivation

Type inference example

```
import java.util.Vector;

class A {

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

Motivation

Type inference example

```
import java.util.Vector;

class A {

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

Result: `m : Vector<T>` $\rightarrow$ `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

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

class A {

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

Result: `m : I<T>` $\rightarrow$ `T` (Structural type).

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:

$\theta \leq^* \theta'$: θ is a subtype of θ'.

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

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

The type unification problem

Definitions:

$\theta \leq^* \theta'$: θ is a subtype of θ'.

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

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

Type unification problem:

For a given set of type term pairs

$$\{ \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \}$$

a unifier (substitution) σ is demanded, such that

$$\sigma(\theta_i) \leq^* \sigma(\theta'_i)$$

Why is type unification not unitary?

This is caused by pairs like:

$$a \lessdot \theta'$$

The results are

$$\{\, a \mapsto \theta \mid \theta \leq^* \theta' \,\}.$$

Type Unification with structural types

Pairs like

$$a \lessdot \theta$$

are themselves results, that means

all subtypes of θ' (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 **TUnify**(C) is given by the following type unification rules application the most often as possible.

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 $T \doteq \theta$, $T \lessdot \theta$, or $\theta \lessdot T$, where T is a type variable and θ is a type term) then C is the result, otherwise the algorithm fails.

Type unification rules

reduce

$$\frac{C \cup \{\, D\text{<}\theta_1, \ldots, \theta_n\text{>} \lessdot D'\text{<}\theta'_1, \ldots, \theta'_n\text{>} \,\}}{C \cup \{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}}$$

where $D\text{<}T_1, \ldots, T_n\text{>} \leq^* D'\text{<}T_1, \ldots, T_n\text{>}$ with T_i are type variables

adapt1

$$\frac{C \cup \{\, D\text{<}\theta_1, \ldots, \theta_n\text{>} \lessdot D'\text{<}\theta'_1, \ldots, \theta'_m\text{>} \,\}}{C \cup \{\, D'\text{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\text{>}[\, T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \doteq D'\text{<}\theta'_1, \ldots, \theta'_m\text{>} \,\}}$$

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

adapt2

$$\frac{C \cup \{\, D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot S_1, S_1 \lessdot S_2, \ldots, S_{k-1} \lessdot S_k, S_k \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \,\}}{\begin{array}{l} C \cup \{\, \sigma(\, D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}\,) \lessdot S_1, S_1 \lessdot S_2, \ldots, S_{k-1} \lessdot S_k, \\ \hfill S_k \lessdot \sigma(\, D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>}\,) \,\} \cup \sigma \end{array}}$$

where

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

erase1 $\dfrac{C \cup \{\, \theta \lessdot \theta' \,\}}{C} \quad \theta \leq^{*} \theta'$

erase2 $\dfrac{C \cup \{\, \theta \doteq \theta' \,\}}{C} \quad \theta = \theta'$

swap $\dfrac{C \cup \{\, \theta \doteq T \,\}}{C \cup \{\, T \doteq \theta \,\}} \quad \theta \notin TV, \;\; T \in TV$

subst $\dfrac{C \cup \{\, T \doteq \theta \,\}}{C[\, T \mapsto \theta] \cup \{\, T \doteq \theta \,\}} \quad \begin{array}{l} T \in TV \text{ and} \\ T \text{ occurs in } C \text{ but not in } \theta \end{array}$

refl $\dfrac{C \cup \{\, \theta \lessdot T_1, T_1 \lessdot T_2, \ldots, T_{n-1} \lessdot T_n, T_n \lessdot \theta \,\}}{C \cup \{\, T_i = \theta \mid 1 \leqslant i \leqslant n \,\}} \quad \theta \notin TV, \;\; T_i \in TV$

Example

```
class A {

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

The result of the type inference algorithm

```
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 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 $\{ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \mathtt{Sub}\texttt{<}\nu_2, \nu_5\texttt{>}, \nu_2 \lessdot \mathtt{Add}\texttt{<}\nu_6, \nu_7\texttt{>} \}$ 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);
    }
}
```

Instance of A

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

```
class Main {

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

The result of TYPE and construct

$$
\begin{aligned}
C = \{\ & \nu_3 \lessdot \nu_5, \\
& \nu_4 \lessdot \nu_7, \\
& \nu_1 \lessdot \texttt{Sub<myInteger}, \nu_5\texttt{>}, \\
& \nu_2 \lessdot \texttt{Add<}\nu_6, \nu_7\texttt{>}, \\
& \texttt{myInteger} \lessdot \nu_1, \\
& \texttt{myInteger} \lessdot \nu_3, \\
& \texttt{myInteger} \lessdot \nu_4\ \}
\end{aligned}
$$

is to be unified.

Unification

$$C = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \texttt{Sub<myInteger}, \nu_5\texttt{>}, \nu_2 \lessdot \texttt{Add<}\nu_6, \nu_7\texttt{>}, \texttt{myInteger} \lessdot \nu_1, \texttt{myInteger} \lessdot \nu_3, \texttt{myInteger} \lessdot \nu_4\ \}$$

Unification

$C = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \texttt{Sub<myInteger}, \nu_5\texttt{>}, \nu_2 \lessdot \texttt{Add<}\nu_6, \nu_7\texttt{>},$
$\texttt{myInteger} \lessdot \nu_1, \texttt{myInteger} \lessdot \nu_3, \texttt{myInteger} \lessdot \nu_4\ \}$

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

$$\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<myInteger}, \texttt{myInteger>},$$
$$\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}.$$

Unification

$C = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \texttt{Sub<myInteger}, \nu_5\texttt{>}, \nu_2 \lessdot \texttt{Add<}\nu_6, \nu_7\texttt{>},$
$\texttt{myInteger} \lessdot \nu_1, \texttt{myInteger} \lessdot \nu_3, \texttt{myInteger} \lessdot \nu_4\ \}$

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

$$\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<myInteger}, \texttt{myInteger>},$$
$$\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}.$$

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

Unification

$C = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \texttt{Sub<myInteger}, \nu_5\texttt{>}, \nu_2 \lessdot \texttt{Add<}\nu_6, \nu_7\texttt{>},$
$\texttt{myInteger} \lessdot \nu_1, \texttt{myInteger} \lessdot \nu_3, \texttt{myInteger} \lessdot \nu_4\ \}$

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

$$\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<myInteger}, \texttt{myInteger>},$$
$$\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}.$$

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

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

Unification

$C = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \texttt{Sub<myInteger}, \nu_5\texttt{>}, \nu_2 \lessdot \texttt{Add<}\nu_6, \nu_7\texttt{>},$
$\texttt{myInteger} \lessdot \nu_1, \texttt{myInteger} \lessdot \nu_3, \texttt{myInteger} \lessdot \nu_4\ \}$

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

$$\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<myInteger}, \texttt{myInteger>}, \\ \nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}.$$

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

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

With the *subst*-rule follows

$$\texttt{myInteger} \lessdot\ \nu_3, \nu_3 \lessdot \texttt{myInteger}, \texttt{myInteger} \lessdot \nu_4, \nu_4 \lessdot \texttt{myInteger}$$

and from this with the *refl*-rule:

$$\nu_3 \doteq \texttt{myInteger}, \nu_4 \doteq \texttt{myInteger}.$$

The result of solve

The result of **solve** is given as:

$$\{\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<myInteger}, \texttt{myInteger>}$$
$$\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}, \nu_6 \doteq \texttt{myInteger},$$
$$\nu_7 \doteq \texttt{myInteger}, \nu_3 \doteq \texttt{myInteger}, \nu_4 \doteq \texttt{myInteger}\}$$

The result of solve

The result of **solve** is given as:

$$\{\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<myInteger,myInteger>}$$
$$\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}, \nu_6 \doteq \texttt{myInteger},$$
$$\nu_7 \doteq \texttt{myInteger}, \nu_3 \doteq \texttt{myInteger}, \nu_4 \doteq \texttt{myInteger}\}$$

```
class Main [myInteger extends ν1,
            ν1 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 σ is a solution of a constraint set C then σ is also a solution of **TUnify**(C).

Soundness and Completeness Theorem

Soundness: If a substitution σ is a solution of a constraint set C then σ is also a solution of **TUnify**(C).

Completeness: Let

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

Then there are substitutions σ'' and σ_{rest}, such that

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

(σ_{rest} is a solution of the remaining pairs $\mathtt{T} \lessdot ty$ and $ty \lessdot \mathtt{T}$)

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

Conclusion

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