

# Java type inference as an Eclipse plugin

Andreas Stadelmeier

1. Oktober 2015

# Typinferenz

- ▶ Typloses Java:

```
class Main{  
    methode(array){  
        array.add("String");  
        return array;  
    }  
}
```

# Typinferenz

- ▶ Typloses Java:

```
class Main{  
    methode(array){  
        array.add("String");  
        return array;  
    }  
}
```

- ▶ Mit inferierten und eingesetzten Typen:

```
class Main{  
    ArrayList<String> methode(ArrayList<String> array){  
        array.add("String");  
        return array;  
    }  
}
```

# Lambda Ausdruck

- ▶ Aufbau:

(Typ1 arg1, ..., TypN argN) -> { [Instruktionen] }

- ▶ Typisierung durch Funktionale Interfaces

- ▶ Beispiel

```
interface Comparable<T>{  
    boolean compareTo(T o);  
}
```

...

```
Comparable<String> compare = (input)-> true;  
Comparable<String> compare2 = (String input)-> {  
    return true; }  
}
```

# FunN-Interface

- ▶ Reihe von Funktionalen Interfaces
- ▶ Nimmt der Typinferenzalgorithmus als gegeben an
- ▶ Für jede natürliche Zahl  $N$  gibt es ein Interface:

```
interface FunN<R, T1, ... , TN>{  
    R apply(T1 arg1, ..., TN argN);  
}
```

- ▶ Für jeden Lambda Ausdruck existiert ein FunN, welches dessen Typ beschreiben kann.

# Motivation

```
class Lambda {  
    op = (m) -> (f) -> {  
        f.apply(this, m);  
        return this;  
    }  
}
```

# Motivation

```
interface Fun1<R, T1>{ R apply(T1 arg1); }
interface Fun2<R, T1, T2>{ R apply(T1 arg1, T2 arg2); }

class Lambda {
  Fun1<Fun1<Lambda, Fun2<Lambda, Lambda, Lambda>>, Lambda>
  op = (m) -> (f) -> {
    f.apply(this, m);
    return this;
  }
}
```

# Ablauf

1. Parsen des Quelltext
2. Generieren von Constraints
3. Unifizierung der Constraints
4. Ermitteln der möglichen Typisierungen
5. Auswahl einer Lösung durch den Benutzer



# Constraints generieren

```
class Lambda{  
  op = (m) -> (f) -> {  
    f.apply(this, m);  
    return this;  
  }  
}
```

# Constraints generieren

```
class Lambda{  
  op = (m) -> (f) -> {  
    f.apply(this, m);  
    return this;  
  }  
}
```

```
(Lambda <. TPH A),  
(TPH F <. Fun2<TPH D, TPH A, TPH M>)  
(Fun1<TPH A, TPH F> <. TPH E)  
(Fun1<TPH E, TPH M> <. TPH op)
```

# Unifizierung

**Unify:** ConstraintsSet  $\rightarrow$  { ConstraintsSet }  $\cup$  { fail }

Input:  $\{ (\theta_1 \triangleleft \theta'_1), \dots, (\theta_n \triangleleft \theta'_n) \}$

Output:  $\{ \sigma_1, \dots, \sigma_m \}$

Post-condition:  $\forall_j \{ (\sigma_j(\theta_1) \leq^* \sigma_j(\theta'_1)), \dots, (\sigma_j(\theta_n) \leq^* \sigma_j(\theta'_n)) \}$   
where  $\leq^*$  is the sub-typing relation

# Unifikatoren

```
op = (m) -> (f) -> {  
    f.apply(this, m);  
    return this;  
}
```

$\sigma(\text{TPH A}) = \text{Lambda}$

$\sigma(\text{TPH F}) = \text{Fun2} \langle \text{TPH D}, \text{TPH A}, \text{TPH M} \rangle$

$\sigma(\text{TPH E}) = \text{Fun1} \langle \text{TPH A}, \text{TPH F} \rangle$

$\sigma(\text{TPH op}) = \text{Fun1} \langle \text{TPH E}, \text{TPH M} \rangle$

# Typen einsetzen

`Fun1<Fun1<Lambda, Fun2<TPH D, Lambda, TPH M>>, TPH M> op = ...`

# Typen einsetzen

```
Fun1<Fun1<Lambda, Fun2<TPH D, Lambda, TPH M>>, TPH M> op = ...
```

```
class Lambda{  
    <D,M> Fun1<Fun1<Lambda, Fun2<D, Lambda, M>>, M> op = ...  
}
```

# Overloading

```
class OL {  
    m(Integer x){ return x + x; }  
    m(Boolean x){ return x || x; }  
}
```

```
class Main{  
    methode(x){  
        ol = new Ol();  
        return ol.m(x);  
    }  
}
```

# Overloading

Typisierungsmöglichkeit 1:

```
class OL {  
    m(Integer x){ return x + x; }  
    m(Boolean x){ return x || x; }  
}
```

```
class Main{  
    Integer methode(Integer x){  
        ol = new OL();  
        return ol.m(x);  
    }  
}
```



# Overloading

Typisierungsmöglichkeit 2:

```
class OL {  
    m(Integer x){ return x + x; }  
    m(Boolean x){ return x || x; }  
}
```

```
class Main{  
    Boolean methode(Boolean x){  
        ol = new OL();  
        return ol.m(x);  
    }  
}
```

# Motivation für Eclipse Plugin

- ▶ Eclipse ist weit verbreitete Java IDE
- ▶ Praktische Nutzung des Typinferenzalgorithmus
- ▶ Problem: Mehrere Lösungen
- ▶ Auswahl einer Typisierung kann Benutzer überlassen werden

# Eclipse Plugin

- ▶ Eclipse ist modular aufgebaut
  - ▶ Bietet explizite Möglichkeiten zur Integration von Plugins
  - ▶ Einzelne Komponenten können ausgetauscht oder erweitert werden
- ▶ Plugin arbeitet auf .jav - Dateien
- ▶ Speichervorgang löst Typinferenzalgorithmus aus
- ▶ Benutzer bestimmt Typisierung bei mehreren Möglichkeiten

# Demo

# Demo

The screenshot shows an IDE window with the following components:

- Menu Bar:** File, Edit, Navigate, Search, Project, Run, Window, Help
- Toolbar:** Standard IDE icons for file operations and navigation.
- Left Panel (Project Explorer):**
  - Project
  - Outline
  - class LambdaTest
    - op (selected)
    - TPH BDU
      - (BDT m) -> {  
return (BDS f) -> {  
f.apply(this, m);  
return this;  
};  
}
    - m
      - TPH BDT
      - LambdaTest

- Main Editor (LambdaTest.java):**

```
class LambdaTest{  
    op = (m) -> (f) -> {f.apply(this,m); return this;};  
}
```
- Bottom Panel (Tasks):**
- Tasks
- 0 items
- Table with columns: Description, Resource, Path
- Status Bar:** Writable, Insert, 5 : 2

# Demo

The screenshot shows an IDE interface with the following components:

- Menu Bar:** File, Edit, Navigate, Search, Project, Run, Window, Help.
- Toolbar:** Includes icons for file operations (New, Open, Save, Print) and navigation (Home, Previous, Next, End).
- Project Explorer (Left):** Shows a project structure with 'class OL', 'class Main', and 'main' (expanded to show '[ol]', 'x', and 'Main').
- Code Editor (Center):** Displays the source code for 'OL.java' and 'Main.java'. The 'main(x)' method in 'Main' is highlighted in blue.

```
class OL {  
    m(x) { return x + x; }  
    m(x) {return x || x; }  
}  
  
class Main {  
    main(x) {  
        ol;  
        ol = new OL();  
        return ol.m(x);  
    }  
}
```
- Tasks Panel (Bottom):** Shows '0 items' and a table with columns: Description, Resource, Path, and L.

# Demo

The screenshot shows an IDE interface with the following components:

- Menu Bar:** File, Edit, Navigate, Search, Project, Run, Window, Help.
- Toolbar:** Includes icons for file operations (New, Open, Save, Print) and navigation (Home, Back, Forward).
- Project Explorer:** Shows a project structure with folders for 'class OL', 'class Main', 'main', '[ol]', 'x', and 'Main'.
- Code Editor:** Displays the source code for 'OL.java' and 'Main.java'.

```
class OL {  
    m(x) { return x + x; }  
    m(x) {return x || x; }  
}  
  
class Main {  
    java.lang.Integer main(x) {  
        ol;  
        ol = new OL();  
        return ol.m(x);  
    }  
}
```
- Tasks Panel:** Shows '0 items' and a table with columns: Description, Resource, Path, and Location.

# Fazit und Ausblick

## Fazit:

- ▶ Globaler Typinferenzalgorithmus für Java 8
- ▶ Generierung von Java Quelltext durch Eclipse Plugin

## Ausblick:

- ▶ Reduzierung der Laufzeit
  - ▶ Entfernen von unnötigen Type Placeholdern und Constraints vor der Unifikation
  - ▶ Verzicht auf Wildcards in FunN-Typen
- ▶ Bytecodegenerierung
- ▶ Intersection Types



# Motivation

```
interface Fun1<R, T1>{ R apply(T1 arg1); }  
interface Fun2<R, T1, T2>{ R apply(T1 arg1, T2 arg2); }  
  
class Lambda {  
  Fun1<? super Fun1<? super Lambda,  
    ? extends Fun2<? super Lambda, ? extends Lambda,  
    ? extends Lambda>>, ? extends Lambda>  
  op = (m) -> (f) -> f.apply(this, m);  
}
```