Java type inference as an Eclipse plugin

Andreas Stadelmeier

1. Oktober 2015

DHBW
Duale Hochschule
Baden-Württemberg

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: $\texttt{ConstraintsSet} \rightarrow \{\, \texttt{ConstraintsSet} \,\} \cup \{\, \mathit{fail} \,\}$

Input: $\{\, (\theta_1 \lessdot \theta'_1), \ldots, (\theta_n \lessdot \theta'_n) \,\}$

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

Post-condition: $\forall_j \{\, (\sigma_j(\,\theta_1\,) \leq^* \sigma_j(\,\theta'_1\,)), \ldots, (\sigma_j(\,\theta_n\,) \leq^* \sigma_j(\,\theta'_n\,)) \,\}$

where $\leq^*$ is the sub-typing relation

Unifikatoren

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

$\sigma(\text{TPH A}) = \text{Lambda}$
$\sigma(\text{TPH F}) = \text{Fun2} < \text{TPH D}, \text{TPH A}, \text{TPH M} >$
$\sigma(\text{TPH E}) = \text{Fun1} < \text{TPH A}, \text{TPH F} >$
$\sigma(\text{TPH op}) = \text{Fun1} < \text{TPH E}, \text{TPH M} >$

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

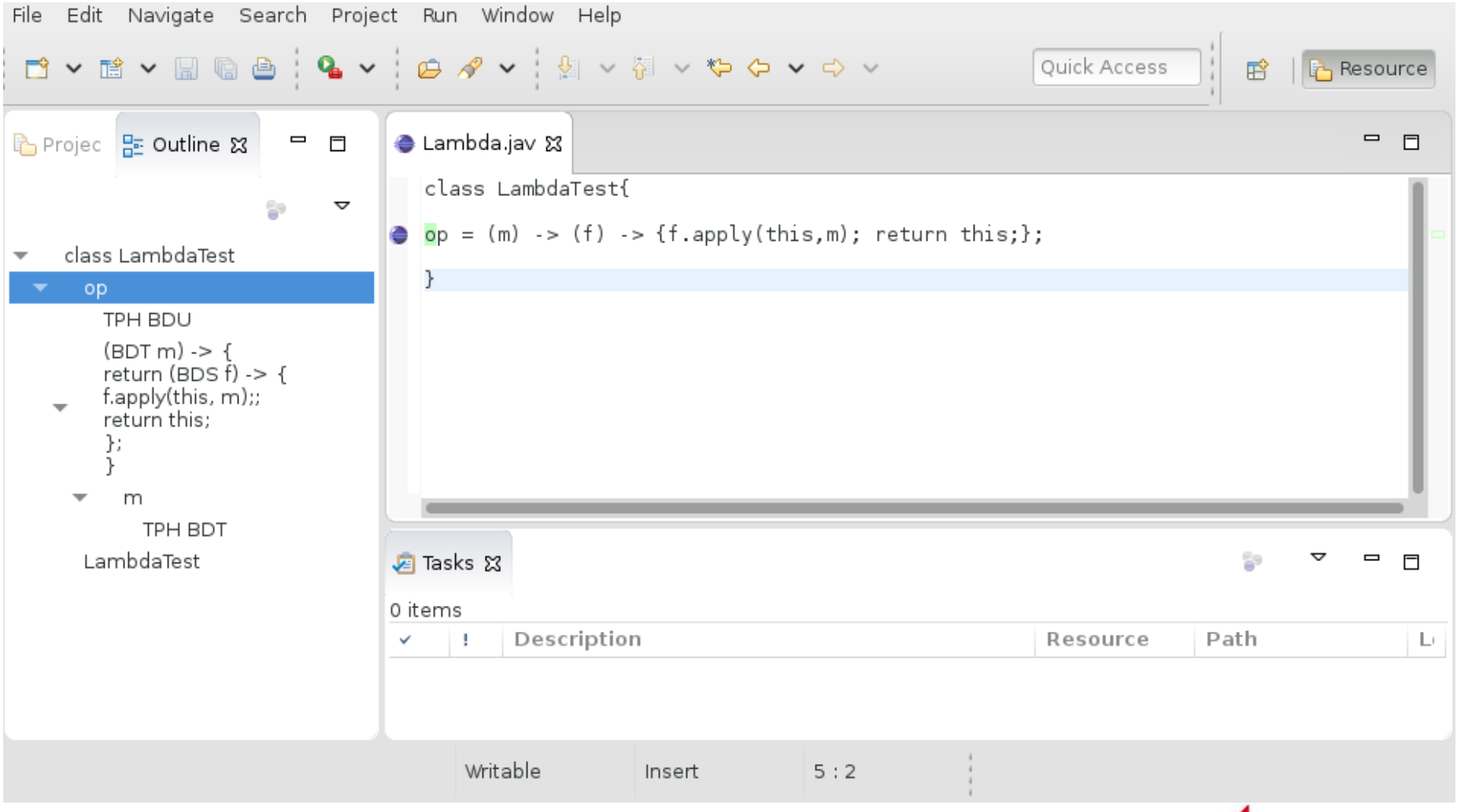
Eclipse Plugin

- 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

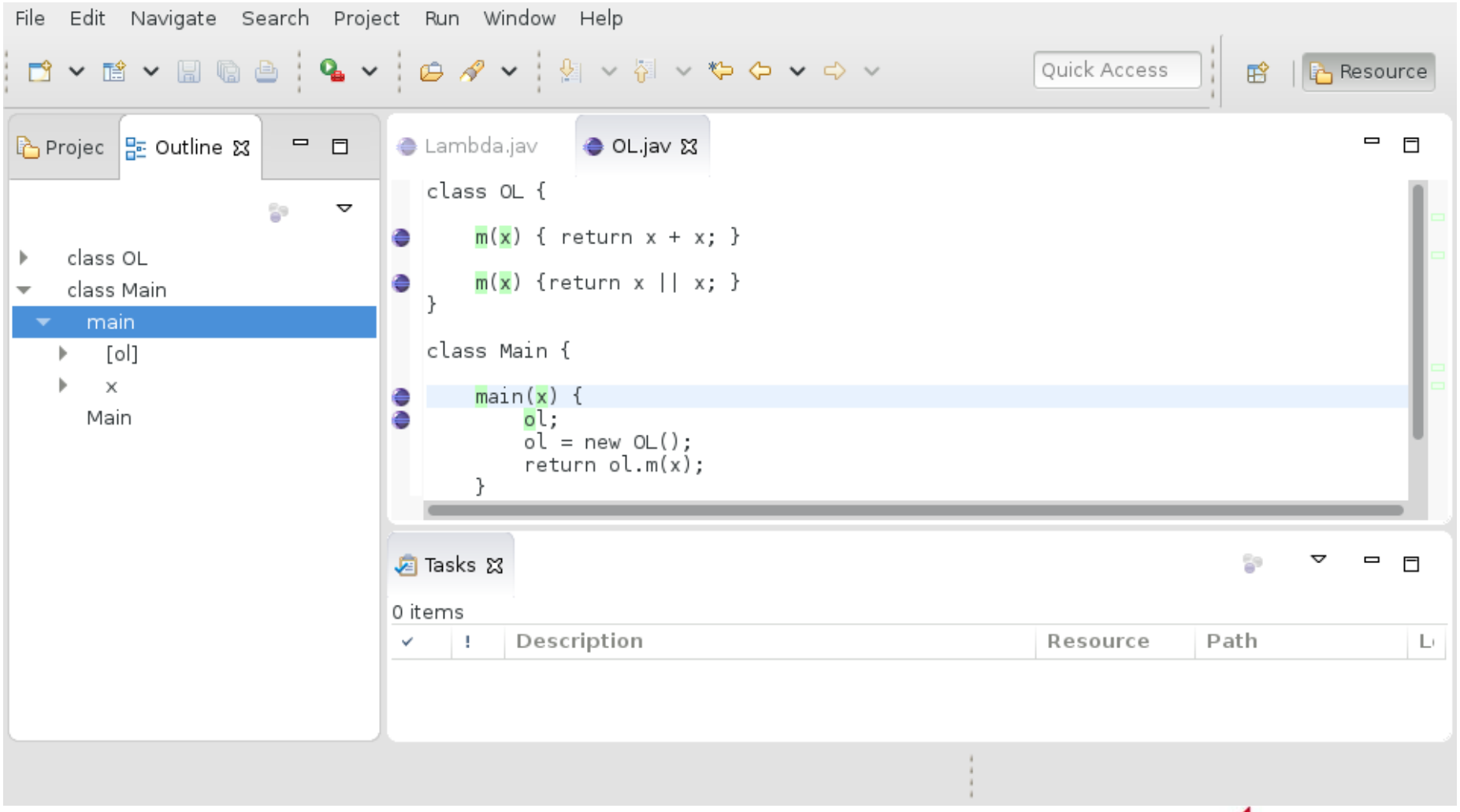
DHBW
Duale Hochschule
Baden-Württemberg

Demo



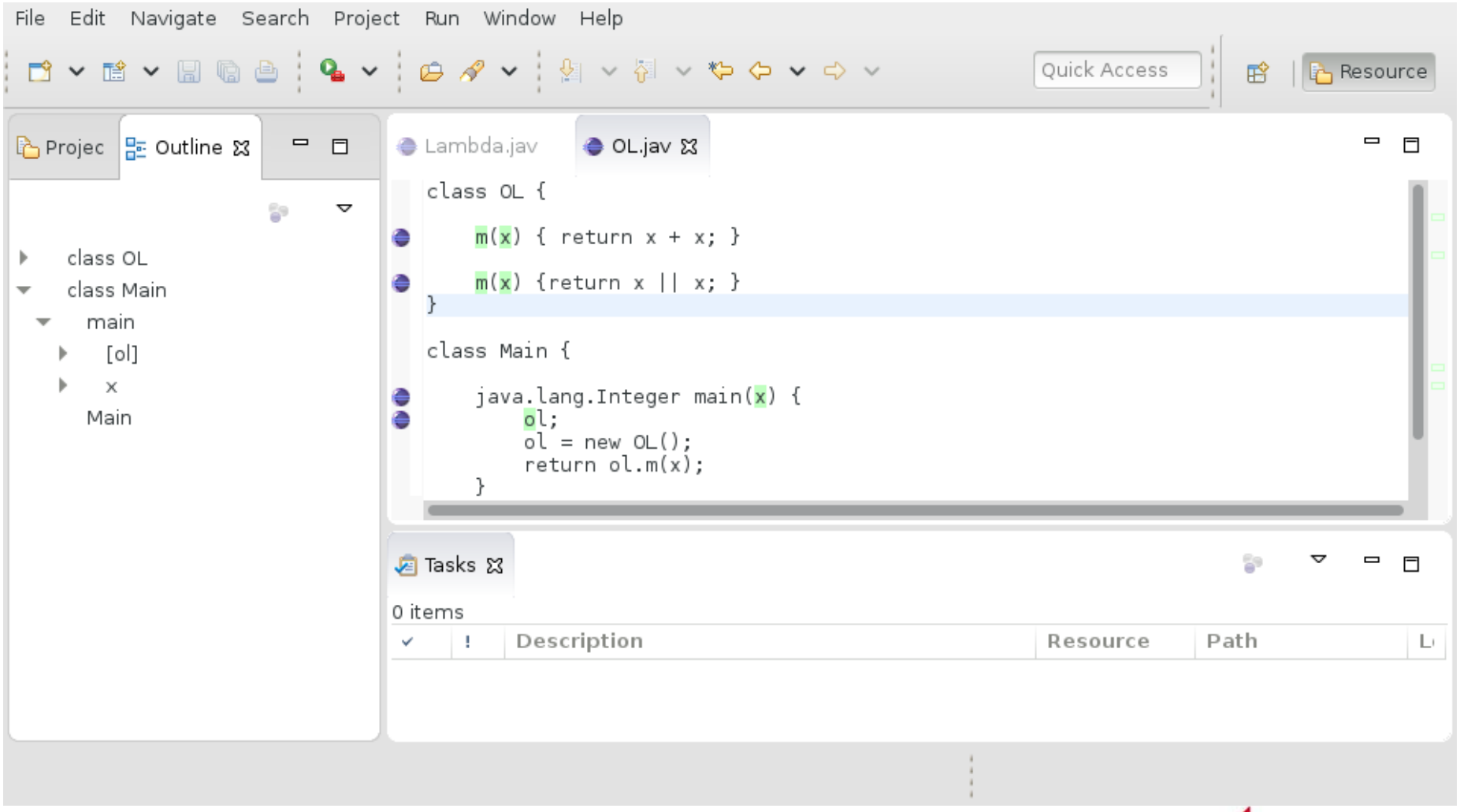
DHBW
Duale Hochschule
Baden-Württemberg

Demo



DHBW
Duale Hochschule
Baden-Württemberg

Demo



DHBW
Duale Hochschule
Baden-Württemberg

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);
}
```