DHBW Stuttgart

# Java-TX: The Language

## Vorlesung Compilerbau

Martin Plümicke

# Overview

# Development of the programming language Java

Many features from functional programming languages are transfered to OO-languages:

**Example Java:**

**Java 5 :** Generics and wildcards

**Java 8 :** lamda-expressions as implementation of functional interfaces (no function types)

**Java 10 :** Completion of *local* type inference

**Java 15 – 22:** Pattern–Matching, Record–Types

# Development of the programming language Java

Many features from functional programming languages are transfered to OO-languages:

**Example Java:**

**Java 5 :** Generics and wildcards

**Java 8 :** lamda-expressions as implementation of functional interfaces (no function types)

**Java 10 :** Completion of *local* type inference

**Java 15 – 22:** Pattern–Matching, Record–Types

**Not yet introduced:**

- Global type inference
- Real function types
- Pattern matching in function headers

# Development of the programming language Java

Many feature from functional programming languages are tranfered to OO-languages:

**Example Java:**

**Java 5 :** Generics and wildcards

**Java 8 :** lamda-expressions as implementation of functional interfaces (no function types)

**Java 10 :** Completion of *local* type inference

**Java 15 – 21:** Pattern–Matching, Record–Types

**Java-TX:**

- Global type inference
- Real function types
- Pattern matching (actual student work)

# Type inference

## First example

```
import java.lang.Integer;

public class Fac {

  getFac(n){
    var res = 1;
    var i = 1;
    while(i<=n) {
      res = res * i;
      i++;
    }
    return res;
  }}
```

# Type inference

## First example

```
import java.lang.Integer

public class Fac {

 java.lang.Integer getFac(java.lang.Integer n){
    var res = 1;
    var i = 1;
    while(i<=n) {
      res = res * i;
      i++;
    }
    return res;
 }}
```

```
class Matrix extends Vector<Vector<Integer>> {
  mul(m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

```
class Matrix extends Vector<Vector<Integer>> {
  mul(m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

Which is the correct type?

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Matrix m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Matrix m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

Is this the *best* type?

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Vector<Vector<Integer>> m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Vector<Vector<Integer>> m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

Which is the *most general* type?

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Vector<? extends Vector<? extends Integer>> m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

# Example Principal Type

## Intersection Type of `mul`

```
mul : Matrix → Matrix
    ∧ Vector<Vector<Integer>> → Matrix
    ∧ Vector<Vector<Integer>> → Vector<Vector<Integer>>
    ∧ Vector<Vector<? extends Integer>> → Matrix
    ∧ Vector<Vector<? extends Integer>>
                                → Vector<Vector<Integer>>
    ∧ Vector<? extends Vector<Integer>> → Matrix
    ∧ Vector<? extends Vector<Integer>>
                                → Vector<Vector<Integer>>
    ∧ Vector<? extends Vector<? extends Integer>> → Matrix
    ∧ Vector<? extends Vector<? extends Integer>>
                                → Vector<Vector<Integer>>
```

# Example Principal Type

## Principal Type of `mul`

```
mul : Matrix → Matrix
    ∧ Vector<Vector<Integer>> → Matrix
    ∧ Vector<Vector<Integer>> → Vector<Vector<Integer>>
    ∧ Vector<Vector<? extends Integer>> → Matrix
    ∧ Vector<Vector<? extends Integer>>
                                → Vector<Vector<Integer>>
    ∧ Vector<? extends Vector<Integer>> → Matrix
    ∧ Vector<? extends Vector<Integer>>
                                → Vector<Vector<Integer>>
    ∧ Vector<? extends Vector<? extends Integer>> → Matrix
    ∧ Vector<? extends Vector<? extends Integer>>
                                → Vector<Vector<Integer>>
```

# Function types

- In Java 8 lambda expressions has fuctional interfaces as target types.
- There are no real function types
- Java-TX has function types as in Scala
- In Java-TX Function types and functional interfaces are integrated

# Simulating function types in Java-8

In the package: `java.util.function` there are

```
public interface Function<T,R> {
  R apply(T t);
}
```

```
public interface BiFunction<T,U,R> {
  R apply(T t, U u);
}
```

# Simulating function types in Java-8

In the package: `java.util.function` there are

```
public interface Function<T,R> {
  R apply(T t);
}
```

```
public interface BiFunction<T,U,R> {
  R apply(T t, U u);
}
```

There are some inconveniences.

# Drawbacks of missing function types

- Missing function types $\Rightarrow$ Introducing `Bi/Function`–Interfaces
- Subtyping problem $\Rightarrow$ Using wildcards
- Impossibility of direct application of lambda expressions
  $\Rightarrow$ Using type-casts

**All problems are solvable, but not pretty!!!**

$\Rightarrow$ **Introducing real function types**

# Introduction of `FunN$$`

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

where

- $\texttt{FunN\$\$<}T'_1,\ldots,T'_N,T_0\texttt{>} \leq^* \texttt{FunN\$\$<}T_1,\ldots,T_N,T'_0\texttt{>}$ iff $T_i \leq^* T'_i$
- In `FunN$$` no wildcards are allowed.

# Introduction of `FunN$$`

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

where

- $\texttt{FunN\$\$<}T'_1,\ldots,T'_N,T_0\texttt{>} \leq^* \texttt{FunN\$\$<}T_1,\ldots,T_N,T'_0\texttt{>}$ iff $T_i \leq^* T'_i$
- In `FunN$$` no wildcards are allowed.

**Lambda–expressions are explicitly typed by Fun*N*$$–types**

```
public class MatrixOP extends Vector<Vector<Integer>> {
  mul = (m1, m2) -> {
    var ret = new MatrixOP();
    var i = 0;
    while(i < m1.size()) {
      var v1 = m1.elementAt(i);
      var v2 = new Vector<Integer>();
      var j = 0;
      while(j < v1.size()) {
        var erg = 0;
        var k = 0;
        while(k < v1.size()) {
          erg = erg + v1.elementAt(k) * m2.elementAt(k).elementAt(j);
          k++; }
        v2.addElement(erg);
        j++; }
      ret.addElement(v2);
      i++;}
    return ret;}
```

# Type of `mul`

- Using `java.util.function.*`:

```
mul: BiFunction<? super Vector<? extends Vector<? extends Integer>>,
                ? super Vector<? extends Vector<? extends Integer>>,
                ? extends MatrixOP>
```

# Type of `mul`

- Using `java.util.function.*`:

```
mul: BiFunction<? super Vector<? extends Vector<? extends Integer>>,
                ? super Vector<? extends Vector<? extends Integer>>,
                ? extends MatrixOP>
```

- Defining an own functional interface MatrixOperation:

```
interface MatrixOperation {
   MatrixOP apply(Vector<? extends Vector<? extends Integer>> arg1,
                  Vector<? extends Vector<? extends Integer>> arg2);
}

mul: MatrixOperation
```

# Type of `mul`

- Using `java.util.function.*`:

```
mul: BiFunction<? super Vector<? extends Vector<? extends Integer>>,
                ? super Vector<? extends Vector<? extends Integer>>,
                ? extends MatrixOP>
```

- Defining an own functional interface MatrixOperation:

```
interface MatrixOperation {
   MatrixOP apply(Vector<? extends Vector<? extends Integer>> arg1,
                  Vector<? extends Vector<? extends Integer>> arg2);
}

mul: MatrixOperation
```

- Java-TX inferred function type:

```
mul: Fun2$$<Vector<? extends Vector<? extends Integer>>,
            Vector<? extends Vector<? extends Integer>>,
            MatrixOP>
```

# Ad-hoc polymorphism

## Example

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

  m(x) { return x || x; }
}

class OLMain {
  main(x) {
    var ol = new OL();
    return ol.m(x);
  }
}
```

# Type of `m`, `m`, and `main`

$$\texttt{m} : \texttt{Integer} \to \texttt{Integer} \\ \wedge\ \texttt{Double} \to \texttt{Double} \\ \wedge\ \texttt{String} \to \texttt{String}$$

$$\texttt{m} : \texttt{Boolean} \to \texttt{Boolean}$$

$$\texttt{main} : \texttt{Integer} \to \texttt{Integer} \\ \wedge\ \texttt{Double} \to \texttt{Double} \\ \wedge\ \texttt{String} \to \texttt{String} \\ \wedge\ \texttt{Boolean} \to \texttt{Boolean}$$

# Generated bytecode

```
> javap OLMain.class
Compiled from "OLMain.jav"
class OLMain {
  public OLMain();
  public java.lang.Integer main(java.lang.Integer);
  public java.lang.Boolean main(java.lang.Boolean);
  public java.lang.String main(java.lang.String);
  public java.lang.Double main(java.lang.Double);
}
```

# Summary and Future work

## Summary

Introducing into Java:

- Global type inference
- Scala function types
- Ad-hoc polymorphism
- Inferred generics

# Summary and Future work

## Summary

Introducing into Java:

- Global type inference
- Scala function types
- Ad-hoc polymorphism
- Inferred generics

## Future work:

- Heterogenous translation
- Pattern matching

# Bachelor/Masterarbeiten–Ausschreibungen I

## IDE-Plugin Schnittstelle für Java-TX mittels Magpie-Bridge

Implementierung einer Schnittstelle zwischen einer Java-IDE (beispliesweise Intellij) und dem Java-TX Compiler:

- Der Java-TX Compiler fungiert als Language-Server
- Magpie-Bridge kann als Framework benutzt werden
- Mithilfe des LSP-Protokolls soll das Tool mit den gängigsten IDE's funktionieren

# Bachelor/Masterarbeiten–Ausschreibungen II

## Portierung auf die Java–Erweiterung Muli

Zur Realisierung von Backtracking eignen sich besonders logische Programmiersprachen. Eine an der Universität Münster entwickelte Java–Erweiterung Muli ergänzt Java um Komponenten von logischen Programmiersprachen.
Ziel der Studienarbeit ist es die Backtracking–Schritte in der bisherigen Implementierung durch logische Konstrukte in Muli zu ersetzen, die das Backtracking automatisieren und damit ggf. erheblich effizienter machen.

# Bachelor/Masterarbeiten–Ausschreibungen III

## Heterogene Übersetzung

Derzeit werden in Java die Generics bei der homogenen Übersetzung nach Bytecode gelöscht (sog. Type Erasure). Dies ist in Java eher selten ein Problem. In Java-TX führt es bei der automatischen Typberechnung dazu, dass berechnete Typen nicht implementiert werden können (z.B. Überladungen). In einer Vorgängerstudienarbeiten wurde die heterogene Übersetzung für die Funktionstypen `FunN$$` realisiert. Nun soll diese auf alle generischen Datentypen erweitert werden. Ein besondere Herausforderung atellt dabei die Vererbung dar. Bei einem Straight-Forward-Ansatz würde dies zu Mehrfachvererbung führen. Dieses Problem soll im Rahmen der Arbeit gelöst und implementiert werden.

# Bachelor/Masterarbeiten–Ausschreibungen IV

## Erweiterung der Typunifikation

Bei der Typunifikation in Java-TX kommen viele Constraints der Form

$$tvar \lessdot Typ'$$

vor. Aktuell werden die Constraints aufgelöst, so dass eine Menge von Lösungen

$$\{\, tvar \doteq Typ \mid Typ \leq^* Typ' \,\}$$

berechnet wird. Dies führt zu einer enorm großer Menge von Lösungen, obwohl in eine Lösung als bounded Generics in Java

$$tvar \texttt{ extends } Typ$$

möglich wäre.
In der Studienarbeit soll die Implementierung der Typunifikation aus einer frühreren Studienarbeit an die vereinfachte Lösung angepasst werden.

# Bachelor/Masterarbeiten–Ausschreibungen V

## Java-TX-Compiler in Java-TX

Im Compilerbau ist ein wichtiger Benchmark, dass ein Compiler in der eigenen Sprache implementiert ist. Dies soll im Rahmen dieser Studienarbeit für Java-TX erfolgen.
Für die Studienarbeit steht der Java-Code des Java-TX-Compilers und die Ergebnisse iner Vorgängerstudienarbeit zur Verfügung. Im Wesentlichen müssen aus dem Code alle Typinformationen entfernt werden und mit dem Java-TX-Compilers compiliert werden.
Sukzessive werden während der Implementierung Fehler gefunden werden. Diese sollten dokumentiert und wenn möglich auch behoben werden.