DHBW Stuttgart

Java-TX

Completing the functional approach in object-oriented languages

Martin Plümicke

Overview

1. Development of OO programming languages
2. Global type inference
3. Function types
4. Additional features
   - Ad-hoc polymorphism
5. Pattern matching
6. Java–TX Compiler Clients
7. Summary and Future work

Development of OO programming languages

Many features from functional programming languages are transfered to OO-languages (e.g. PIZZA, Scala, Java, C#, C++).

Main challenges

- States in imperative OO languages
- Subtyping
- Overloading and overriding

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Type inference: *Local* type inference

No type inference for method declarations und recursive lamda expressions

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Type inference: *Local* type inference

No type inference for method declarations und recursive lamda expressions

Functions as first-class citizens: Lamda-expressions as implementation of functional interfaces

No function types

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Type inference: *Local* type inference

No type inference for method declarations und recursive lamda expressions

Functions as first-class citizens: Lamda-expressions as implementation of functional interfaces

No function types

Algebraic data types, pattern–matching: Record–Types and Pattern–Matching (`switch`- and `instanceof`-statement)

No patterns in method headers

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Type inference: *Local* type inference

No type inference for method declarations und recursive lamda expressions

Functions as first-class citizens: Lamda-expressions as implementation of functional interfaces

No function types

Algebraic data types, pattern–matching: Record–Types and Pattern–Matching (`switch`- and `instanceof`-statement)

No patterns in method headers

⇒ In Java-TX we address these gaps

Global 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; } }
```

Principal type

Principal type for a functional program:

A type-scheme for a declaration is a principal type-scheme, if any other type-scheme for the declaration is a generic instance of it. [Damas, Milner 1982]

Principal type

Principal type for a functional program:

A type-scheme for a declaration is a principal type-scheme, if any other type-scheme for the declaration is a generic instance of it. [Damas, Milner 1982]

Principal type for Java-TX:

An intersection type-scheme with minimal number of elements for a declaration is a principal type-scheme, if any other type-scheme for the declaration is a subtype of generic instance of one element of the intersection type-scheme.

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.

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

$$\texttt{Function<}T'_1, T_0\texttt{>} \not\leq^* \texttt{Function<}T_1, T'_0\texttt{>}, \quad \text{for } T_i \not\leq^* T'_i$$

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

$$\texttt{Function<}T'_1, T_0\texttt{>} \not\leq^* \texttt{Function<}T_1, T'_0\texttt{>}, \quad \text{for } T_i \leq^* T'_i$$

Example:
For `Integer` $\leq^*$ `Number` $\leq^*$ `Object` holds:

$$\texttt{Number} \to \texttt{Number} \leq^* \texttt{Integer} \to \texttt{Object}$$

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

$$\texttt{Function<}T'_1, T_0\texttt{>} \not\leq^* \texttt{Function<}T_1, T'_0\texttt{>}, \quad \text{for } T_i \not\leq^* T'_i$$

Example:
For `Integer` $\leq^*$ `Number` $\leq^*$ `Object` holds:

$$\texttt{Number} \to \texttt{Number} \leq^* \texttt{Integer} \to \texttt{Object}$$

but

```
Function<Number,Number> f_NumNum = ...
Function<Integer,Object> f_IntObj = f_NumNum
```

is **wrong!**, as

$$\texttt{Function<Number,Number>} \not\leq^* \texttt{Function<Integer,Object>}$$

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

$$\texttt{Function<}T'_1, T_0\texttt{>} \not\leq^* \texttt{Function<}T_1, T'_0\texttt{>}, \quad \text{for } T_i \not\leq^* T'_i$$

Example:
For `Integer` $\leq^*$ `Number` $\leq^*$ `Object` holds:

$$\texttt{Number} \to \texttt{Number} \leq^* \texttt{Integer} \to \texttt{Object}$$

but

```
Function<Number,Number> f_NumNum = ...
Function<Integer,Object> f_IntObj = f_NumNum
```

is **wrong!**, as

$$\texttt{Function<Number,Number>} \not\leq^* \texttt{Function<Integer,Object>}$$

Subtyping of functional interfaces could be realized by wildcards.

Subtyping II

It holds

$$\texttt{Function<}T_1'\texttt{,}T_0\texttt{>} \leq^* \texttt{Function<?\,super\,T}_1\texttt{,?\,extends\,T}_0'\texttt{>}, \quad \text{for } T_i \leq^* T_i'.$$

This means

```
Function<? super Integer,? extends Object> f_IntObj =
f_NumNum
```

is correct.

Large example for function types

```
//A -> (B -> (((A, B) -> C) -> C)))
```

```
g = x -> y -> f -> f.apply(x,y);
```

Large example for function types

```
//A -> (B -> (((A, B) -> C) -> C)))
Func.<? super A,
      ? extends Func.<? super B,
                      ? extends Func.<? super BiFunc.<? super A,
                                                      ? super B,
                                                      ? ext. C>>,
                                      ? extends C>>>
g = x -> y -> f -> f.apply(x,y);
```

Large example for function types

```
//A -> (B -> (((A, B) -> C) -> C)))
Func.<? super A,
      ? extends Func.<? super B,
                      ? extends Func.<? super BiFunc.<? super A,
                                                      ? super B,
                                                      ? ext. C>>,
                                      ? extends C>>>
g = x -> y -> f -> f.apply(x,y);
```

Ugly syntax!

Direct application of lambda expressions

In the λ-calculus β-conversion (direct application of a lambda expression to its arguments) is possible:

$$(\lambda\, x.E)arg = E[x/arg].$$

In Java 8 this lambda term would have the following form:

```
(x -> h(x)).apply(arg);
```

Such expressions are not permitted, as the lambda expression has no type.

Direct application of lambda expressions

In the λ-calculus β-conversion (direct application of a lambda expression to its arguments) is possible:

$$(\lambda\, x.E)arg = E[x/arg].$$

In Java 8 this lambda term would have the following form:

```
(x -> h(x)).apply(arg);
```

Such expressions are not permitted, as the lambda expression has no type.

```
((Function<T,R>)x -> h(x)).apply(arg);
```

is **ok!**, as there is cast–expression.

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 with integration into the concept of target 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 Fun*N*$$

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

where

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

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

Example function types

```
//A -> (B -> (((A, B) -> C) -> C)))
Func.<? super A,
      ? extends Func.<? super B,
                      ? extends Func.<? super BiFunc.<? super A,
                                                      ? super B,
                                                      ? ext. C>>,
                                                      ? extends C>>>


g = x -> y -> f -> f.apply(x,y);
```

Example function types

```
//A -> (B -> (((A, B) -> C) -> C)))
Func.<? super A,
      ? extends Func.<? super B,
                      ? extends Func.<? super BiFunc.<? super A,
                                                      ? super B,
                                                      ? ext. C>>,
                                                      ? extends C>>
Fun1$$<A, Fun1$$<B, Fun2$$<A, B, C>, C>>
g = x -> y -> f -> f.apply(x,y);
```

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

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

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

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

Pattern matching I

```
sealed interface List<A> permits Cons, Nil {}

record Cons<A>(A a, List<A> l) implements List<A> {}

record Nil<A>()implements List<A>  {}

void m(List<A> x) {
  switch(x) {
    case Cons(A a, Cons(A b, List l))
                    -> System.out.println(a);
    case Cons(A a, Nil()) -> System.out.println(a);
    case Nil() -> System.out.println("Nil");
  }
}
```

Pattern matching II

```
sealed interface List<A> permits Cons, Nil {}

record Cons<A>(A a, List<A> l) implements List<A> {}

record Nil<A>()implements List<A>  {}

void m(List<A> x) {
  switch(x) {
    case Cons(var a, Cons(var b, var l))
                    -> System.out.println(a);
    case Cons(var a, Nil()) -> System.out.println(a);
    case Nil() -> System.out.println("Nil");
  }
}
```

Pattern matching in Java-TX I

```
sealed interface List<A> permits Cons, Nil {}

record Cons<A>(A a, List<A> l) implements List<A> {}

record Nil<A>()implements List<A>  {}

void m(List<A> x) {
  switch(x) {
    case Cons(a, Cons(b, l)) -> System.out.println(a);
    case Cons(a, Nil()) -> System.out.println(a);
    case Nil() -> System.out.println("Nil");
  }
}
```

Pattern matching in Java-TX II

```
sealed interface List<A> permits Cons, Nil {}

record Cons<A>(A a, List<A> l) implements List<A> {}

record Nil<A>()implements List<A>  {}

void m(Cons(a, Cons(b, l))) { System.out.println(a); }
void m(Cons(a, Nil())) { System.out.println(a); }
void m(Nil()) { System.out.println("Nil"); }
```

Java-TX Compiler Clients

Based on LSP:

1. VS Code
2. Intellij
3. Emacs

Live Demo I

fst.jav | Faculty.jav | MainFaculty.jav | List.jav 4 | OL.j | Mainfst.jav | MainOL.jav

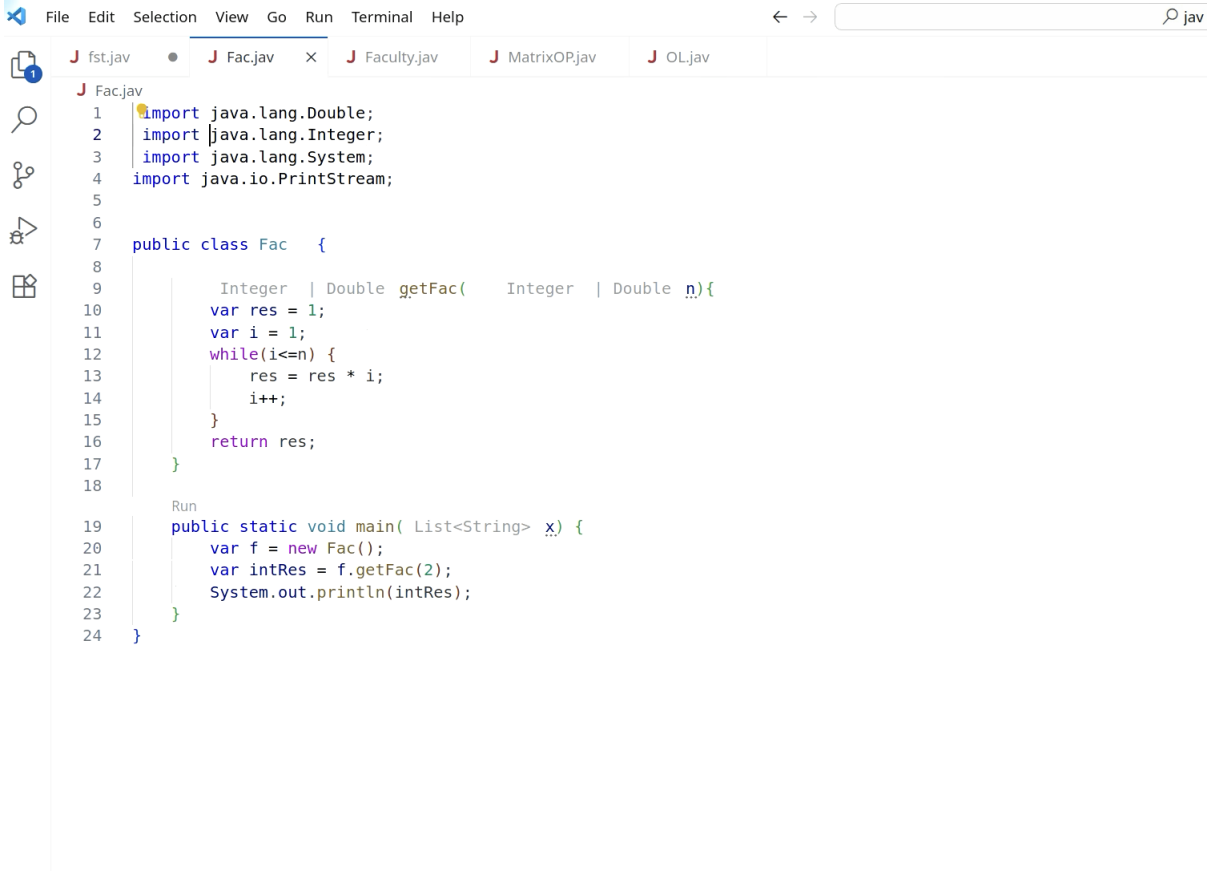
Users > pl > ResearchPapers > KPS_25 > Ruben_Martin > jav > fst.jav

```
1
2
3   class fst {
4       id(x) {
5           return x;
6       }
7   }
```

Users > pl > ResearchPapers > KPS_25 > Ruben_Martin > jav > Mainfst.jav

```
1   import java.lang.System;
2   import java.io.PrintStream;
3   import fst;
4   import java.lang.String;
5
6   public class Main {
        Run
7       public  static main(x) {
8           fst f = new fst();
9           var res = f.id("hallo");
10          System.out.println(res);
11
12      }
13  }
```

Live Demo II


File Edit Selection View Go Run Terminal Help
jav
fst.jav
Fac.jav
Faculty.jav
MatrixOP.jav
OL.jav
Fac.jav
1 import java.lang.Double;
2 import java.lang.Integer;
3 import java.lang.System;
4 import java.io.PrintStream;
5
6
7 public class Fac {
8
9 Integer | Double getFac(Integer | Double n){
10 var res = 1;
11 var i = 1;
12 while(i<=n) {
13 res = res * i;
14 i++;
15 }
16 return res;
17 }
18
Run
19 public static void main(List<String> x) {
20 var f = new Fac();
21 var intRes = f.getFac(2);
22 System.out.println(intRes);
23 }
24 }

Live Demo III

Users > pl > ResearchPapers > KPS_25 > Ruben_Martin > jav > Faculty.jav

```
3   import java.lang.System;
4   import java.io.PrintStream;
5
6
7   public class Faculty {
8       public   Fun1$$<Integer, Integer>  | Fun1$$<Double,… fact = (x) -> {
9               if (x == 1) {
10                  return 1;
11              }
12              else {
13                return x * (fact.apply(x-1));
14              }
15          };
16
17      public  Integer  | Double  getFact( Integer  | Double  x) {
18          return fact.apply(x);
19      }
20
21      public  void  main() {
22          var f = new Faculty();
23          var intRes = f.getFact(3);
24          System.out.println(intRes);
25      }
26  }
```

Live Demo IV

Users > pl > ResearchPapers > KPS_25 > Ruben_Martin > jav > J OL.jav

```
1  import java.lang.String;
2  import java.lang.Integer;
3  import java.lang.Double;
4  import java.lang.Boolean;
5
6  public class OL1  {
7
8      String  | Integer  | Double m1( String  | Int… x) { return x + x; }
9
10
11     Boolean m1( Boolean x) { return x || x; }
12
13
14 }
15
16 public class OL2 {
17
18     String  | Boolean  | Integer  | Double m2( St… x) {
19         var ol;
20         ol = new OL1();
21         return ol.m1(x);
22     }
23
24
25 }
```

Users > pl > ResearchPapers > KPS_25 > Ruben_Martin > jav > J MainOL.jav

```
1  import java.lang.String ;
2  import java.lang.System;
3  import java.io.PrintStream;
4  import java.lang.Float;
5  import OL2;
6
7  class MainOL {
       Run
8      public  static main(x) {
9          var ol = new OL2();
10
11         //the function m2 is applied to an integer
12         var res1 = ol.m2(2);
13         System.out.println(res1);
14         //the m2 is applied to a double
15         var res2 = ol.m2(2.0);
16         System.out.println(res2);
17         //the m2 is applied to a double
18         var res3 = ol.m2("Hallo");
19         System.out.println(res3);
20     }
21 }
```

Live Demo V

Users > pl > ResearchPapers > KPS_25 > Ruben_Martin > jav > J MatrixOP.jav

```
8
9   public class MatrixOP    extends Vector<Vector<Integer>> {
10
11      public MatrixOP () {
12      }
13
14      public MatrixOP(vv) {
15          Integer i;
16          i = 0;
17          while(i < vv.size()) {
18            this.add(vv.elementAt(i));
19            i=i+1;
20        }
21      }
22
23      mul = (m1, m2) -> {
24          var ret = new MatrixOP();
25          var i = 0;
26          while(i < m1.size()) {
27              var v1 = m1.elementAt(i);
28              var v2 = new Vector<Integer>();
29              var j = 0;
30              while(j < v1.size()) {
31                  var erg = 0;
32                  var k = 0;
33                  while(k < v1.size()) {
34                      erg = erg + v1.elementAt(k)
35                                * m2.elementAt(k).elementAt(j);
36                      k++; }
37  //                v2.addElement(new Integer(erg));
38                  v2.addElement(erg);
39                  j++; }
40              ret.addElement(v2);
41              i++;
42          }
43          return ret;
44      };
```

Users > pl > ResearchPapers > KPS_25 > Ruben_Martin > jav > J MatrixOP.jav

```
37  //                v2.addElement(new Integer(erg));
43          return ret;
44      };
45
        Run
46      public static main(x) {
47          var vv = new Vector<Vector<Integer>>();
48          var v1 = new Vector<Integer> ();
49          v1.addElement(2);
50          v1.addElement(2);
51          var v2 = new Vector<Integer> ();
52          v2.addElement(3);
53          v2.addElement(3);
54
55          //MatrixOP
56          var mOp1 = new MatrixOP();
57          mOp1.addElement(v1);
58          mOp1.addElement(v2);
59          MatrixOP mOp2 = mOp1.mul.apply(mOp1, mOp1);
60          System.out.print(mOp1); System.out.print(" *
61      }
62  }
63
```

Summary and Future work

Summary

Introducing into Java:

- Global type inference
- Scala function types
- Ad-hoc polymorphism
- Java-TX Compiler Clients

Summary and Future work

Summary

Introducing into Java:

- Global type inference
- Scala function types
- Ad-hoc polymorphism
- Java-TX Compiler Clients

Future work:

- Implementing pattern matching
- Avoiding capture-conversion