Java-TX: The Language

Martin Plümicke
Duale Hochschule Baden-Württemberg, Campus Horb Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract. Java-TX is an extension of Java. The main new features are global type inference and real function types for lambda expressions. These extensions lead to a new more powerful overloading mechanism, which means that the principal type of a method is an intersection of method types.

1 Introduction

The programming language Java is extended since version 1.5 by many features from functional programming languages. In version 1.5 generics are introduced. Generics are known as parametric polymorphism in functional programming languages. In contrast to functional programming languages as Haskell or SML object-oriented languages like Java allows subtyping and states of objects. Therefore the variance of type arguments has to be defined. In PIZZA, the first approach of parametric polymorphism in Java-like languages, the arguments were declared as invariant, which means for Vector<T> ≤ Collection<T> and Integer ≤ Object holds Vector<Integer> ≤ Collection<Integer> but neither Vector<Integer> ≤ Collection<Object> (covariance) nor Vector<Object> ≤ Collection<Integer> (contravariance) is correct. Invariance of type arguments is a hard restriction. Therefore in Java 5.0 so-called wildcards were introduced. Wildcards allow in some cases covariance and contravariance, respectively.

In Java 8 lambda expressions were introduced, but no function types. The types of lambda expressions are defined as target types, which are functional interfaces (essentially interfaces with one method).

Local type inference was introduced in the versions five, seven, and nine. In Java 5.0 the automatic determination of parameter instance was introduced. In Java 7 the diamond operator was introduced. In Java 9, finally, the var keyword for types of local variables was introduced.

The features global type inference (no type declarations are necessary without losing static typing), real function types, and pattern-matching are not addressed in Java since now.

\( \leq^* \) stands for the subtyping relation.
In Java-TX we address global type inference and real function types.

The paper is organized as follows: First we describe in Section 2 the global type inference in Java-TX. After that in Section 3 we describe the introduction of real function types. In Section 4 we describe two additional features, a powerful overloading mechanism and the inference of generics, which follow both from the type inference algorithm. In the last section we conclude and give an outlook.

2 Globale type inference for Java

Global type inference allows to leave out all type annotations. Similar as in functional programming languages like Haskell the compiler determines a principal typing, such that Java-TX is statically typed as original Java [Plü14]. Let us consider first a simple example.

Example 1. The following program is given:

```java
import java.lang.Integer;

public class Fac {

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

It is the simple iterativ implementation of the faculty function. The return and the argument type of getFac are leaved out. The type inference algorithm has to infer the types. The types are determined by the declaration in line 6 and the overloaded operator * in line 9. To reduce the complexity of the type inference algorithm for overloaded operator just as overloaded methods only types are inferred which are explicitly imported by the keyword import. Therefore for getFac the typing

```
java.lang.Integer getFac(java.lang.Integer n)
```

is inferred.

Let us consider a second more complex example.

Example 2. The following program is given:
import java.util.Vector;

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;
    }
}

The class Matrix is implemented as an extension of Vector<Vector<Integer>>. The method mul implements the multiplication of two matrices. The obvious typing would be

Matrix mul(Matrix m)

The question is, if this typing is the only possible typing. If not then it is the question if it is the best typing.

It is simple to see that there are other correct typings. E.g.

Matrix mul(Vector<Vector<Integer>> m)

and

Vector<Vector<Integer>> mul(Vector<Vector<Integer>> m)

is correct. But

Matrix mul(Vector<? extends Vector<? extends Integer>> m)

is also correct.

If we summarize all correct typings of mul to an intersection the type of mul is given in Fig. [1]

We remember the usual subtyping definition for function types.

Definition 1 (Subtyping relation $\leq^*$ on function types). For two given functions types $(\tau_1, \ldots, \tau_n) \to \tau$ and $(\theta_1, \ldots, \theta_n) \to \theta_0$ holds

$$(\tau_1, \ldots, \tau_n) \to \tau \leq^* (\theta_1, \ldots, \theta_n) \to \theta_0$$

if $\theta_i \leq^* \tau_i$ and $\tau \leq^* \theta$. 
Fig. 1. Intersection type of mul

Regarding this definition we should define the intersection type of all maximal elements as the best type.
In the above example \( \text{Vector}\langle ? \text{ extends Vector}\langle ? \text{ extends Integer}\rangle \rangle \rightarrow \text{Matrix} \) is the supertype of all other types. Therefore this should be the best type of \( \text{mul} \).

In addition, we consider the declaration of a principal type for functional programs [DMS2]: 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.

Finally we combine the ideas of best types in Java with the principal type of functional programs and give a definition for a principal type of Java methods:

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.

In the following we give the formal definition.

**Definition 2 (Principal types of Java methods).** An intersection type of a method \( m \), which contains no overloaded call (for overloading cp. Section 4.1)

\[
\begin{align*}
m : & ((\theta_{1,1}, \ldots, \theta_{1,n}) \rightarrow \theta_1) \\
& \& \ldots \\
& ((\theta_{m,1}, \ldots, \theta_{m,n}) \rightarrow \theta_m)
\end{align*}
\]

is called principal if for any correct type annotated method declaration

\[
\text{rty} \ m(ty1 \ a1, \ldots, tyn \ an) \{ \ldots \}
\]

there is an element

\[
((\theta_{i,1}, \ldots, \theta_{i,n}) \rightarrow \theta_i)
\]

of the intersection type and there is a substitution \( \sigma \), such that
\[
\sigma(\theta_i) \leq^* \sigma(\theta_{i,1}), \ldots, \sigma(\theta_{i,n}) \leq^* rty,  ty_1 \leq^* \sigma(\theta_{i,1}), \ldots, ty_n \leq^* \sigma(\theta_{i,n})
\]

and the number of elements of the intersection type is minimal.

The corresponding type inference algorithm we gave in [Plü14] and the underlying unification is given in [Plü09,SP18].

3 Real function types

Indeed Java allows lambda expressions, but there are no function types. Instead there are functional interface as target types of lambda expressions. There are many disadvantages because of the missing of function types. Java-TX solves this disadvantages by introducing function types similar as in Scala without loosing the convenience of functional interfaces as target type of lambda expressions [PS17].

In Java 8 function types are simulated in the package: java.util.function:

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

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

There are some inconveniences.

Subtyping

Although for subtypes holds (cp. Def. [1])

\[(T'_1, \ldots, T'_N) \rightarrow T'_0 \leq^* (T_1, \ldots, T_N) \rightarrow T'_0, \quad \text{iff} \ T_i \leq^* T'_i\]

for the functional interface Function

\[\text{Function}<T'_1, T'_0> \leq^* \text{Function}<T_1, T'_0>, \quad \text{for} \ T_i \leq^* T'_i,\]

is not correct, as Java has use-side variance. Therefore arguments of types without wildcards are invariant.

Example 3. For Integer \leq^* Number \leq^* Object holds:

\[\text{Number} \rightarrow \text{Number} \leq^* \text{Integer} \rightarrow \text{Object}\]

but

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

is wrong!, as
This problem could be solved by wildcards. It holds

\[
\text{Function}\langle T_1',T_0\rangle \leq^* \text{Function}\langle ?\text{ super } T_1, ?\text{ extends } T_0'\rangle, \text{ for } T_i \leq^* T_i'.
\]

This means

\[
\text{Function}\langle ?\text{ super } \text{Integer}, ?\text{ extends } \text{Object}\rangle \ f_{\text{IntObj}} = f_{\text{NumNum}}
\]
is correct.

**Direct application of lambda expressions**

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

\[
(\lambda x.E)\text{arg} = E[x/\text{arg}].
\]

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

\[
(x \to h(x)).\text{apply}(\text{arg});
\]

Such expressions are not permitted. As the lambda expression has no explicit type it is not obvious if the method \(\text{apply}\) exists, at all. This problem could be solved by introducing a type-cast.

\[
((\text{Function}\langle T,R\rangle)x \to h(x)).\text{apply}(\text{arg});
\]

**Summary**

The Drawbacks of missing function types are all solved in Java 8:

**Missing function types:** The function types are replaced by the functional interfaces \(\text{Bi/Function}\) in the package \(\text{java.util.function}\).

**Subtyping problem:** The problem that the functional interface’s behaviour differ from the usual definition of subtyping is solved by using wildcards.

**Impossibility of direct application of lambda expressions:** The impossibility to apply an lambda expression directly to its arguments is solved by using type-casts.

This means all problems are solvable, but the solution are not pretty. Therefore we introduced real function types in Java-TX. We extended Java by two sets of special functional interfaces

```java
interface FunN$$\langle-T1, \ldots , -TN , +R\rangle \{ 
    R apply(T1 arg1, \ldots , TN argN);
}

interface FunVoidN$$\langle-T1, \ldots , -TN\rangle \{
    void apply(T1 arg1, \ldots , TN argN);
}
```
where

- \( \text{Fun} \langle T_1', \ldots, T_k', T_0 \rangle \leq^* \text{Fun} \langle T_1, \ldots, T_k, T_0 \rangle \) iff \( T_i \leq^* T_i' \)
- In \( \text{Fun} \langle \_ \rangle \) no wildcards are allowed.

The Lambda–expressions are explicitly typed by \( \text{Fun} \langle \_ \rangle \)-types.

**Example 4.** Let us considering the following changed matrix program:

```java
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;
    }
}
```

In Java 8 there are two possibilities to type the field `mul`:

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

  ```java
  ```

  This type declaration is less readable. Especially the mixture of super- and extends-wildcards in the second argument are very curious.

- Defining an own functional interface `MatrixOperation`:

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

  `mul: MatrixOperation`

  This type declaration is very short, but the type `MatrixOperation` hides the most informations.
In contrast, Java-TX infers the function type:

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

This type is complex, too. But the arguments of the function type `Fun2$` have no wildcards. This reduces the confusion.

4 Additional features

There are some positiv side-effects from the type inference. We will consider two of them in this section. The first is a powerful overloading mechanism and the second is the possibility to generalize type variables from the result of the type inference algorithm.

4.1 Overloading

The idea The following example shows the overloading mechanism.

```
Example 5. Let the classes OL and OLMain be given.

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

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

The type of the first method `m` is:

```
  m : Integer → Integer
& Double → Double
& String → String,
```

as `+` is an overloaded operation symbol. The second method `m` has the type

```
  m : Boolean → Boolean.
```

In class OLMain an instance of OL is created and on the instance the overloaded method `m` is called. This means that `main` has all four types of the both methods `m`:

```
main : Integer → Integer
& Double → Double
& String → String
& Boolean → Boolean
```
Extended principal type definition In Definition 2 principal types are defined restricted to non overloaded calls. For this let us consider the following example.

Example 6. Let the following Java program be given:

```java
import java.util.Vector;
import java.util.Stack;

class Put {
    <T> putElement(T ele, Vector<T> v) {
        v.addElement(ele);
    }

    <T> putElement(T ele, Stack<T> s) {
        s.push(ele);
    }

    main(ele, x) {
        putElement(ele, x);
    }
}
```

The inferred intersection type of `main` is

```
main: T × Vector<T> → void & T × Stack<T> → void.
```

With Def. 2 the principal would be `T × Vector<T> → void`. This would not be the principal type as the stack's application would disappear. Therefore we have extend the definition.

Definition 3 (Principal type of Java methods with overloading). An intersection type of a method \( m \)

```
\begin{align*}
m &: ((\theta_{i,1}, \ldots, \theta_{i,n}) &\rightarrow \theta_i) \\
&\& \&
&\& \& ((\theta_{m,1}, \ldots, \theta_{m,n},) &\rightarrow \theta_m)
\end{align*}
```

is called principal if the number of elements of the intersection is minimal and for any correct type annotated method declaration

```
\begin{align*}
\text{rty } m(ty1 \ a1, \ldots, tyn \ an) \ {\ldots}
\end{align*}
```

there is an element \(((\theta_{i,1}, \ldots, \theta_{i,n},) &\rightarrow \theta_i)\) of the intersection type and there is a substitution \( \sigma \), such that

```
\sigma(\theta_i) \leq^* \text{rty}, ty1 \leq^* \sigma(\theta_{i,1}), \ldots, tyn \leq^* \sigma(\theta_{i,n})
```

and the call-graphs of \((\theta_{i,1}, \ldots, \theta_{i,n},) &\rightarrow \theta_i\) and \((ty1, \ldots, tyn) &\rightarrow \text{rty}\) are equal.
Example 7. Continuing Example 6, the principal type of \texttt{main} is
\[
\texttt{main}: \mathbb{T} \times \text{Vector}<\mathbb{T}> \rightarrow \text{void} \land \mathbb{T} \times \text{Stack}<\mathbb{T}> \rightarrow \text{void},
\]
as the call graphs of \(\mathbb{T} \times \text{Vector}<\mathbb{T}> \rightarrow \text{void}\) and \(\mathbb{T} \times \text{Stack}<\mathbb{T}> \rightarrow \text{void}\) differ.

A detailed regard to call graphs and the corresponding algorithm can be found in \cite{Plu08}.

**Heterogenous translation**

Example 8. Let us consider the following \texttt{Java} program:

```java
class OLFun {
    m(f, x) {
        x = f.apply(x+x);
        return x;
    }
}
```

We get for the arguments of \texttt{m} the following typings:

- \(\text{Fun1}$$<\mathbb{D}, \mathbb{D}> \times \mathbb{D} \rightarrow \mathbb{D}\)
- \(\text{Fun1}$$<\mathbb{I}, \mathbb{I}> \times \mathbb{I} \rightarrow \mathbb{I}\)
- \(\text{Fun1}$$<\mathbb{S}, \mathbb{S}> \times \mathbb{S} \rightarrow \mathbb{S}\).

In \texttt{Java–bytecode} the arguments of the generic types are not considered (\textit{type-erasure}). Indeed in bytecode the arguments are contained, but they are used only for the typecheck. The JVM considers only the descriptions. For the class \texttt{OLFun} the method headers in bytecode looks like this:

```java
public java.lang.Double m(Fun1$$<java.lang.Double, java.lang.Double>, java.lang.Double);
    descriptor: (LFun1$$;Ljava/lang/Double;)Ljava/lang/Double;
public java.lang.Integer m(Fun1$$<java.lang.Integer, java.lang.Integer>, java.lang.Integer);
    descriptor: (LFun1$$;Ljava/lang/Integer;)Ljava/lang/Integer;
public java.lang.String m(Fun1$$<java.lang.String, java.lang.String>, java.lang.String);
    descriptor: (LFun1$$;Ljava/lang/String;)Ljava/lang/String;
```

This overloading is similar as in Example 5, no problem, as the method call can be resolved by the second argument.

But if we erases the second argument

```java
class OLFun {
    x;

    m(f) {
        x = f.apply(x+x);
        return x;
    }
}
```
the type-erasure is a problem, as the method headers in bytecode are:

```java
public java.lang.Double m(Fun1$$<java.lang.Double, java.lang.Double>);
descriptor: (LFun1$$;)Ljava/lang/Double;
public java.lang.Integer m(Fun1$$<java.lang.Integer, java.lang.Integer>);
descriptor: (LFun1$$;)Ljava/lang/Integer;
public java.lang.String m(Fun1$$<java.lang.String, java.lang.String>);
descriptor: (LFun1$$;)Ljava/lang/String;
}
```

Now no method resolving is possible as all three methods have the same argument `Fun1$$`. This problem could only be solved by heterogeneous translations, which preserves the arguments in the descriptors. In [ORW00] an idea for heterogeneous translation was given.

### 4.2 Generalized type variables

Similar as in type inference of functional programming languages, free type variables, which are not instanced by other types after type inference are generalized to generics. In comparison to functional programming languages in Java subtyping leads to a more powerful generalizations mechanism.

**Example 9.** Let us consider the identity function.

```java
class Id {
  id(x) { return x; }
}
```

What is the type of `id`? In an approach without subtyping the result would be

```
<T> T id (T x)
```

The result of our type inference algorithm is: For the following typing by fresh type variables (type-placeholders)

```java
class Id { K id(L b)({ return (b)::L; })::M}
```

the result is \{ \{ (L < K) \}\}. This means that the following class is generated in bytecode:

```java
class Id {
  <L extends K, K> K id (L x) { return x; }
}
```

Let us consider an example with a field declared as a lambda expression.

**Example 10.** Let the identity function be given as a lambda expression assigned to a field of a class:

```java
class lambdaId {
  lambdaId = x -> x;
}
```
The result of the type inference algorithm is given as

```java
class lambdaId {
    K lambdaId = ((x)::CIC -> ((x)::CIC)::L)::P;
}
```

with

\{
\{K = Fun1$$<CIC, CID>), (P = Fun1$$<L, BPP>),
    (CIC < L), (L < BPP), (BPP < CID)}\}

In this example the free type variables are not restricted to a method as in Example 8. The type variables are valid in the whole class, as the field `lambdaId` is valid in the whole class, too. Therefore the free type variables become generics of the class. This means the following class is generated in bytecode:

```java
class lambdaId<CIC extends CID, CID> {
    Fun1$$CIC, CID> lambdaId = x -> x;
}
```

5 Implementation

A prototypical compiler for Java-TX has been implemented. The compiler is written in Java, itself. Additionally, an eclipse plugin is presented, which allows to use the full convenience of Java type inference. (https://www.hb.dhbw-stuttgart.de/javatx)

6 Summary and Outlook

Java has been developed in the last year such that concepts from the functional programming languages have been introduced. In the paper we presented an extension of Java, called Java-TX. Java-TX continues the range of introducing functional programming language features into Java. We added the feature of global type inference. Global type inference means that Java programs can be written without any type annotation. Java-TX preserves static typing by a type inference algorithm, which infers a principal type for all fields and methods.

We discussed the principal type property in Java with and without overloading. The second main extension of Java-TX is the introduction of real function types. We gave some examples which shows that the lack of real function is very harmful. We introduced Scala–like function types. For lambda expressions we defined these function types as explicit types. At once we preserved the concept of target typing for functional interfaces.

Finally we presented two side-effects from the extensions. First, we showed that one function declaration can be overloaded by different types. Second, we presented the generalization concept for free type variables, which is more powerful than in functional programming languages.
We showed that the overloading mechanism has some restrictions, caused by the type-erasure (erasure of argument types in parameterized types during compilation). Therefore we plan to realize an approach of heterogeneous translation. There are some approaches. One approach is given in \textsc{PIZZA} [\textsc{ORW00}]. A further approach is used be \textsc{C#} [\textsc{C#17}]. There are possibilities following the ideas of [\textsc{UTO13}].

Another feature from functional programming languages which had not been introduced into Java is pattern matching. There is an approach in \textsc{PIZZA} [\textsc{OW97}], too. This approach is restricted to switch-case statements. In combination with type inference more flexible approaches could be possible.

\textbf{References}

\textsc{C#17}. C# language specification, 2017.


\textsc{PS17}. Martin Plümicke and Andreas Stadelmeier. Introducing Scala-like function types into Java-TX. In \textit{Proceedings of the 14th International Conference on Managed Languages and Runtimes, ManLang 2017}, pages 23–34, New York, NY, USA, 2017. ACM.
