Java Type System – Proposals for Java 10 or 11

Martin Plümicke
Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract. In this paper we will present ideas for the extension of the Java type system. On the one hand Java could get real function types. There are some disadvantages of the Java 8 approach to use target types as types for lambda expressions. In our approach the idea of target typing is preserved but extended by real function types. From this extension follows an extension of our type inference algorithm.
On the other hand we extend the Java type system by intersection types of function types. The principal types of functions in Java are in general intersection types.

Introduction

The development of Java in the last decade has introduced many features from functional programming languages. While in Java 5.0 [GJSB05] generics are introduced in Java 8 [GJS+14] lambda expressions are added. In [Plü07,Plü15] we proposed Java type inference systems that allows to give Java programs without type annotations. Type inference systems are also well-known from functional programming languages.
All these three approaches have some difficulties but were good enough. We address these difficulties in this paper. For this we extend the Java type system again. We call the language Java Type Extended (Java-TX), that is a conservative extension of Java 8.
In Java 8 lambda expressions themselves have no explicit types. They get as target types so-called functional interfaces (interfaces with one method) from the context. This approach has the advantage that many implementations of existing call-back interfaces are improved. But it has also some disadvantages i.e. the subtyping property. Therefore in Java-TX we add a concept of real function types as explicit types of lambda expressions. For this we define a set of special interfaces FunN*, that represent real function types. We address this extension in Section 1.
In Section 2 we explain the role of the FunN*–types in our type inference system. The inferred types of Java functions are in general intersections of function types. As Java allows no intersection types, the intersections had to be resolved by the programmer. Since now, we do this by an eclipse plugin [Sta15]. In Java-TX
we introduce intersection types of function types. In Section 3 this extension is addressed.

Finally, we close with a conclusion and give an outlook.

1 Real function types

In the past we considered two different type inference algorithms for lambda expressions. While in [Plü11] real function types are considered, in [Plü15] the Java 8-like functional interface are used. In Java-TX we merge these both approaches, as both have some advantages.

1.1 The special interface FunN*

A lambda expression in Java 8 has no explicit type. The type is determined by the compiler from the context in which the expression appears. This means that one lambda expression can have different types in different contexts.

```java
Callable<String> c = () -> "done";
PrivilegedAction<String> a = () -> "done";
```

In the first context for the lambda expression the type `Callable<String>` is determined, while in the second context `PrivilegedAction<String>` is determined.

In [Plü14] we summarized all functional interfaces to equivalence classes, which single abstract method’s have the same typings. As a representation of the respective classes we introduce for simulating function types a predefined collection of interfaces for all \( N \in \mathbb{N} \):

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

The following example shows the inconvenience of this approach.

Example 1. Let be the following function \( g \) defined:

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

The curried function \( g \) takes three arguments, where the third argument is a function, that is applied to the first and the second argument. In a functional programming language a principal type of \( g \) would be

\[
A \rightarrow (B \rightarrow (((A, B) \rightarrow C) \rightarrow C)).
\]

But with the FunN-construction the equivalent type would be

\[
\]
\[
? super A>}
\]
Nearly no programmer would give \( g \) such type, although it is the principal type. In Java-TX we extend these interfaces to special interfaces \( \text{Fun}N^* \), where the subtyping property is changed in comparison to Java. The special interfaces \( \text{Fun}N^* \) correspond to functions types in Scala \([Ode14]\).

The language Java-TX contains interfaces for all \( N \in \mathbb{N} \)

\[
\begin{align*}
\text{interface Fun}_N^*_{<+R,-T_1, \ldots, -T_N>} & \{
\text{R} \ \text{apply}(T_1 \ \text{arg}_1, \ldots, T_N \ \text{arg}_N); \\
\}
\end{align*}
\]

where \( \text{Fun}_N^*_{<T_0,T_1, \ldots, T_N>} \leq^* \text{Fun}_N^*_{<T'_0,T'_1, \ldots, T'_N>} \) iff \( T_i \leq^* T'_i \) with \( \leq^* \) as subtyping relation. For \( \text{Fun}N^* \) no wildcards are allowed.

Let us consider the following example

\[
\begin{align*}
\text{Object} \ m(\text{Integer} \ x, \text{Fun}_1^*_{<\text{Object}, \text{Integer}>} f) & \{
\text{return} \ f.\text{apply}(x); \\
\}
\end{align*}
\]

It is obvious, that the following application is correct:

\[
\begin{align*}
\text{Fun}_1^*_{<\text{Object}, \text{Integer}>} f_{\text{IntObj}} = \ldots \\
\text{Object} \ x_2 = m(2, f_{\text{IntObj}});
\end{align*}
\]

But for \( \text{Integer} \leq^* \text{Number} \leq^* \text{Object} \) also

\[
\begin{align*}
\text{Fun}_1^*_{<\text{Integer}, \text{Integer}>} f_{\text{IntInt}} = \ldots \\
\text{Object} \ x_1 = m(2, f_{\text{IntInt}});
\end{align*}
\]

is correct, as \( \text{Fun}_1^*_{<\text{Integer}, \text{Integer}>} \) is a subtype of \( \text{Fun}_1^*_{<\text{Object}, \text{Integer}>} \)

and

\[
\begin{align*}
\text{Fun}_1^*_{<\text{Number}, \text{Number}>} f_{\text{NumNum}} = \ldots \\
\text{Object} \ x_3 = m(2, f_{\text{NumNum}});
\end{align*}
\]

is correct, as \( \text{Fun}_1^*_{<\text{Number}, \text{Number}>} \) is also a subtype of \( \text{Fun}_1^*_{<\text{Object}, \text{Integer}>} \).

**Example 2.** Considering again Example 1 the program

\[
\begin{align*}
g = x \to y \to f \to f.\text{apply}(x,y);
\end{align*}
\]

has in Java-TX the type \( \text{Fun}_1^*_{<\text{Fun}_1^*_{<\text{Fun}_1^*_{<C,Fun2^*_{<C,A,B>,B>},A>}}}} \).

1.2 \( \text{Fun}_N^* \) as types of methods

We can also give \( \text{Fun}_N^* \)–types to methods. This means with the class CL

class CL {
    T_0 \text{meth} (T_1 \ \text{x}_1, \ldots, T_N \ \text{x}_N) \{ \ldots \}
}

the method reference \( \text{CL::meth} \) has the type \( \text{Fun}_N^*_{<T_0,T_1, \ldots, T_N>} \).

The advantage of this definition is that method references can be used as lambda expression. Also subtyping and direct applications work in the same manner.

\(^1\) The arguments are covariant resp. contravariant, written as in Scala [Ode14]
1.3 Integration of real function types into Java-8

We preserve in our approach the great benefits of the target typing in Java 8 by integration both concepts. The target typing is extended in the following way:

- A lambda expression itself has an explicit FunN*-type.
- A lambda expression fits any target type, which must be a functional interface, if its method's type in FunN*-representation is a supertype of the explicit type.

Example 3. Let us consider again:

```java
Callable<String> c = () -> "done";
PrivilegedAction<String> a = () -> "done";
```

The explicit type of the lambda expressions () -> "done" is Fun0*<String>. The types of the methods call of Callable<String> and run of PrivilegedAction<String> have also the type Fun0*<String>. This means that the target types are compatible.

2 Type inference

Another feature well-known from functional programming languages is type inference. In object-oriented languages, type inference is only in the restricted form of local type inference [PT98] implemented, while in Java 8 some elements are introduced. It is possible to leave out the argument types of lambda expression (instead (ty a) -> expr it is possible to write (a) -> expr). Furthermore the so-called diamond operator is introduced. This means that it is possible to write new Class<> and the parameters of Class are inferred.

But complete type inference, especially type inference of recursive declared functions is not implemented.

The main reason for this lack is that the results in the defined Java type system are generally not unique.

We address this problem in different approaches. In [Plü07] we gave a type inference algorithm for Java with generics including wildcards. In [Plü11] we presented a type inference algorithm for Java with real function types. In [Plü15] finally we presented a type inference algorithm for Java with lambda expressions and functional interfaces.

In this section we present the type inference algorithm for Java-TX. For this we have to combine the approaches of type inference for real function types [Plü11] and type inference for functional interfaces [Plü15]. Java-TX uses the special interfaces FunN* for function types, that are nominal types. Therefore we use the base of [Plü15]. The differences in the results are solved by adapting the underlying type unification [Plü09].
2.1 The algorithm

The type inference algorithm (Figure 1) takes a set of type assumptions TypeAssumptions and a untyped class Class and gives a pair of a set of remaining constraints Constraints and a typed class TClass.

\[
TI: \text{TypeAssumptions} \times \text{Class} \rightarrow \{(\text{Constraints}, \text{TClass})\}
\]

\[
TI(\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), \text{fdecls})) = \\
\text{let } (\text{Class}(\tau, \text{extends}(\tau'), \text{fdecls}), \text{ConS}) = \\
\text{TYPE}(\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), \text{fdecls})) \\
\{ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n) \} = \text{SOLVE}(\text{ConS}) \\
\text{in } \{(cs_i, \sigma_i(\text{Class}(\tau, \text{extends}(\tau'), \text{fdecls})))) | 1 \leq i \leq n \}
\]

Fig. 1. The type inference algorithm

TI consists of two main functions TYPE and SOLVE, where TYPE inserts type annotations, widely type variables as placeholders, in the Java class and determines a set of type constraints and SOLVE solves the constraints by our type unification algorithm [Plü09]. The result of SOLVE is a set of pairs \{ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n) \}, where the \(cs_i\) consists of remaining constraints (\(a \preceq a'\)) of types variables and \(\sigma_i\) consists of solutions (\(a = \theta\)), where (\(a \preceq a'\)) means \(a\) has to be a subtype of \(a'\) and (\(a = b\)) means \(a\) and \(b\) are equal.

Let us consider the class Matrix in Figure 2. A class Matrix is declared as an extension of Vector<Vector<Integer>>. \text{op} is a function defined by a lambda expression in curried representation with two arguments. First it takes a matrix and second it takes a function, that has as arguments two matrices and returns another matrix. The result of \text{op} is the application of the function (second argument) to its object (this) and its first argument. The method \text{mul} is the ordinary matrix multiplication in lambda representation. Finally, in \text{main} the function \text{op} is applied. The \text{op}-function of matrix \text{m1} is applied to the matrix \text{m2} and the function \text{mul} of \text{m1}. In the figure the class Matrix is shown in Java 8 and in Java-TX. The Java-TX program shows the possibilities to declare programs without type annotations. A little curious is the declaration of local variables \text{ret}; \text{v1}; \text{v2}; \text{m1}; and \text{m2};. This is necessary as for the reason of unambiguousness Java-TX retains the Java property that all variables must be declared before used.

2.2 Type unification

In the function SOLVE the type unification is called to solve the type constraints. In [Plü09] we described the type unification for the Java type system. The introduction of the FunN* types induces an extension of this unification. The three most important added unifications rule are given in Figure 3. In the rules \(a \preceq b\) means \(a\) must be a subtype of \(b\) and \(a = b\) means \(a\) and \(b\) must be equal.
//Java 8 with type annotations
class Matrix extends Vector<Vector<Integer>> {

Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
op = (Matrix m) -> (Fun2<Matrix, Matrix,Matrix> f) -> f.apply(this, m);

Fun2<Matrix, Matrix,Matrix> mul = (Matrix m1, Matrix m2) -> {
    Matrix ret = new Matrix();
    for(int i = 0; i < size(); i++) {
        Vector<Integer> v1 = m1.elementAt(i);
        Vector<Integer> v2 = new Vector<Integer>();
        for (int j = 0; j < size(); j++) {
            int erg = 0;
            for (int k = 0; k < v1.size(); k++) {
                erg = erg + v1.elementAt(k).intValue() * (m2.elementAt(k)).elementAt(j).intValue(); }
            v2.addElement(erg); }
        ret.addElement(v2); }
    return ret; }

public static void main(String[] args) {
    Matrix m1 = new Matrix(...);
    Matrix m2 = new Matrix(...);
    (m1.op.apply(m2)).apply(m1.mul);}
}

//Java-TX without type annotations
class Matrix extends Vector<Vector<Integer>> {

op = (m) -> (f) -> f.apply(this, m);

mul = (m1, m2) -> {
    ret; ret = new Matrix();
    for(int i = 0; i < size(); i++) {
        v1; v1 = m1.elementAt(i);
        v2; v2 = new Vector<Integer>();
        for (int j = 0; j < size(); j++) {
            int erg = 0;
            for (int k = 0; k < v1.size(); k++) {
                erg = erg + v1.elementAt(k).intValue() * (m2.elementAt(k)).elementAt(j).intValue(); }
            v2.addElement(erg); }
        ret.addElement(v2); }
    return ret; }

public static void main(String[] args) {
    m1; m1 = new Matrix(...);
    m2; m2 = new Matrix(...);
    (m1.op.apply(m2)).apply(m1.mul);}
}

Fig. 2. Matrix in Java 8 respectively in Java-TX without th and type annotations
The rule \textbf{reduceFunN} describes the reduction of the FunN* interfaces. This means that the parameters are in covariant respectively contravariant relations. The rules \textbf{greaterFunN} and \textbf{smallerFunN} describes the solutions of all greater respectively all smaller FunN*-types. This means that the parameters of the FunN*-types gets greater respectively smaller.

\subsection*{2.3 Example}

In the following we show the functionality of the type inference algorithm \textbf{TI} by the application to the function \textit{op} from Figure 2. First the function \textbf{TYPE} is called, that inserts type annotations, widely type variables as placeholders, and determines a set of type constraints. The abstract syntax of the program with type annotations inserted is:

\begin{verbatim}
op::aop = (\langle m:a_m \rangle \mapsto ((f:a_f) \mapsto f.apply(this:Matrix, m:a_m):a_3) :Fun1*<\lambda f, a_m>) :Fun1*<a_f, a_m>
\end{verbatim}

and the set of constraints is given as:

\begin{verbatim}
\{ (Fun1*<a_f, a_m> \leq a_op), (Fun1*<a_app, a_f> \leq a_f),
(a_f \simeq Fun2*<a_3, a_1, a_2>, (Matrix \leq a_1), (a_m \leq a_2),
(a_3 \leq a_app) \}
\end{verbatim}

With applying \textbf{greaterFunN} to Fun1*<a_f, a_m> \leq a_op) we get

\begin{verbatim}
\{ (a_op \simeq Fun1*<b_1>), (a_f \leq b'), (b_1 \leq a_m) \}
\end{verbatim}

With applying \textbf{greaterFunN} to Fun1*<a_app, a_f> \leq a_f we get

\begin{verbatim}
\{ (a_f \simeq Fun1*<c', c_1>), (a_app \leq c'), (c_1 \leq a_f) \}
\end{verbatim}

With substituting a_f in a_f \leq b' and again applying \textbf{greaterFunN} we get

\begin{verbatim}
\{ (b' \simeq Fun1*<d', d_1>), (c' \leq d'), (d_1 \leq c_1) \}
\end{verbatim}
With substituting \( a_f \) in \( c_1 \triangleq a_f \) and applying \( \text{smallerFunN*} \) we get

\[
\{ (c_1 \triangleq \text{Fun2*}<x,x'_1,x'_2>), (x \triangleq a_3), (a_1 \triangleq x'_1), (a_2 \triangleq x'_2) \}
\]

With substituting \( c_1 \) in \( d_1 \triangleq c_1 \) and applying \( \text{smallerFunN*} \) again we get

\[
\{ (d_1 \triangleq \text{Fun2*}<y,y'_1,y'_2>), (y \triangleq x), (x'_1 \triangleq y'_1), (x'_2 \triangleq y'_2) \}
\]

This leads to the following set of constraints (considering only the relevant constraints):

\[
\begin{align*}
\{ \text{Matrix} \triangleq a_1 \triangleq x'_1 \triangleq y'_1, \\
b_1 \triangleq a_m \triangleq a_2 \triangleq x'_2 \triangleq y'_2, \\
y \triangleq x \triangleq a_3 \triangleq a_{app} \triangleq c' \triangleq d', \\
a_{op} \triangleq \text{Fun1*<Fun1*<d', Fun2*<y,y'_1,y'_2>>, b_1>}, \\
a_{xf} \triangleq \text{Fun1*<c', Fun2*<x,x'_1,x'_2>>,} \\
a_f \triangleq \text{Fun2*<a_3,a_1,a_2>} \}
\end{align*}
\]

The result of \( \text{SOLVE} \) (considering only the relevant constraints and solutions) is given as following set of pairs:

\[
\begin{align*}
\{ (b_1 \triangleq a_2 \triangleq x'_2 \triangleq y'_2), \\
y \triangleq x \triangleq a_3 \triangleq c' \triangleq d' \}, \\
\{ a_{op} \triangleq \text{Fun1*<Fun1*<d', Fun2*<y,y'_1,y'_2>>, b_1>}, \\
a_{xf} \triangleq \text{Fun1*<c', Fun2*<x,x'_1,x'_2>>,} \\
a_f \triangleq \text{Fun2*<a_3,a_1,a_2>} \}) \\
| \text{Matrix} \triangleq a_1 \triangleq x'_1 \triangleq y'_1 \}
\end{align*}
\]

The result of \( \text{TI} \) is given as the application of the \( \text{SOLVE} \)'s results to the result program of \( \text{TYPE} \). The result consists of a set of typings for \( \text{op} \):

```java
class Matrix extends Vector<Vector<Integer>> {
    <y2', b1 extends y2', d', y extends d'>
    Fun1*<Fun1*<d', Fun2*<y, X, y2'>>, b1>
    op = (m) -> (f) -> f.apply(this, m);
```

where \( \text{Matrix} \triangleq \ast X \).

If we compare this result with the Java 8 program in Figure 2 we see that the types are more general: On the one hand argument and result types are type variables and on the other hand there are more than one principal results (\( \text{Matrix} \triangleq \ast X \)).

This example shows that the results of the type inference algorithm are not unique in general. The reason is, that the type unification algorithm has multiple results.
Let us consider another example. In Figure 4 we show how the type inference algorithm deals with overloading. The result of the type inference for the method `main` is:

\[
\{ X \text{ main}(X \ a) \ |
\begin{align*}
X \ &= \text{OL} \\
OL \ &= \text{new OL()} \\
\text{return OL.m(a);}
\end{align*}
\} \quad X \in \{ \text{Integer, String, Long, Double, Boolean, Float} \}
\]

In this example the property of multiple results is induced by the overloading of the operators `+` and `||`, while in `Matrix` the property is induced by the property that the type unification has multiple results.

Upto now, we have had a simple but practical solution to resolve multiple results. We have had an eclipse plugin [Sta15] as user interface such that the user can select the desired solution.

In this paper we consider a new approach, that resolves multiple solutions by extending the **Java** type system by intersections of function types.

### 3 Intersection function types

In this section we extend the **Java** type system by introducing intersections of function types. In [Plü08] we considered this for **Java** without FunN*-types. Now we extend the idea to function types.

Let us look again on the class `Matrix` from Section 2. A first approach to define an intersection type could be to introduce for each supertype of `Matrix` an element (Figure 5). This definition makes less sense, as there are many subtype relations

\[\]
between elements of the intersection. Therefore a better approach would be to define the type of \( \text{op} \) as the intersection of all maximal elements in the subtyping ordering. Then the type of \( \text{op} \) would be as given in Figure 6.

In general a principal type should be defined. The idea of principal typing is, that if an expression has multiple types, there is one type, from which all other types are derivable. This type is called the principal type. E.g. in [DM82] a principal type for functional programs is defined, where the possibility to derive is the generic instantiation of type variables. E.g. the identity function has the principal type \( \text{id}: \text{a} \to \text{a} \), where \( \text{a} \) is a type variable. This means all other types of \( \text{id} \) are instantiations of \( \text{a} \to \text{a} \), e.g. \( \text{id}: \text{int} \to \text{int} \) or \( \text{id}: \text{char} \to \text{char} \).

In [vB93] a generalization of this definition is given, that replaces the generic instantiation by an arbitrary derive-function.

We define for Java-TX the following principal typing:

**Definition 1 (Java-TX principal typing).** An intersection type with minimal number of elements of an expression is a principal type, if any (non-intersection) type of the expression is a subtype of a generic instance of one element of the intersection type and the call-graphs are identical.

For the explanation of this definition we give three further examples. We extend the matrix example by introducing a parameter for \( \text{Matrix}<\text{E}> \) and an additional class \( \text{intMatrix} \), that contains the method \( \text{mul} \) (cp. Figure 7).

The type of \( \text{op} \) applied in the method \( \text{main} \) is

\[
\text{Fun1} \ast \text{<Fun1}<\text{intMatrix}, \text{Fun2}<\text{intMatrix},\text{intMatrix}, \text{intMatrix}>>\,\text{, intMatrix}>>
\]

The corresponding element of the principal intersection type is

\[
\text{Fun1} \ast \text{<Fun1}<\text{d'}, \text{Fun2}<\text{y},\text{Vector}<\text{? extends Vector}<\text{? extends E}>, \text{y2'}>>\,\text{, b1}>
\]
**Fig. 7.** Parametrized Matrix

Let us consider again the class OL in Figure 4. The principal type of main is:

\[
\text{main} : \text{Integer} \to \text{Integer} \& \text{String} \to \text{String} \& \text{Long} \to \text{Long} \& \text{Double} \to \text{Double} \& \text{Boolean} \to \text{Boolean} \& \text{Float} \to \text{Float}
\]

Finally we give an example that shows why the call-graph must be considered. Let the class Put in Figure 8 be given.

**Fig. 8.** The class Put

The principal type of main is:

\[
\text{main} : T \times \text{Vector}<T> \to \text{void} \& T \times \text{Stack}<T> \to \text{void}.
\]
If the call-graph would not be considered, \( T \times \text{Stack}<T> \rightarrow \text{void} \) would not belong to the principal type, as \text{Stack} is a subtype of \text{Vector}. But this type is necessary as \text{main} defines different functions on \text{Vector} and \text{Stack}.

If we compare the matrix example with the others, we recognize, that the matrix example uses the lambda expression representation for functions, while in \text{OL} and \text{Put} methods are used. The Java-TX type systems allows for both representations intersection types.

4 Conclusion and outlook

4.1 Conclusion

We have presented an extension of the Java type system. On the one hand we proposed to introduce real function types. We gave an approach similar to the approach in Scala. We showed how both concepts, the concept of using functional interfaces as target types for lambda expressions, as well as our concept of real function types, can be integrated. So the advantages of both concepts can be used.

We showed the necessary extension of our type inference algorithm to use real function types.

On the other hand we have introduced function intersection types, that are in general results of our type inference algorithm.

4.2 Outlook

For the implementation of both the real function types and the intersection types generics in byte-code are necessary. In [ORW00] two ways to compile PIZZA [OW97] (an early Java extension with generics) are given. Beside the common homogenous compilation (type-erasures) there is given an approach of heterogenous compilation, which preserves the type parameters. This approach is designed for JVM version \(< 5\). This approach has to be redesigned and adopted to version 8.

References


