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 expression 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 `Fun`N`*`, that represent real function types. We address this extension in Section 1.
In Section 2 we explain the role of the `Fun`N`*`–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 Fun$N$∗

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.

```
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 $N \in \mathbb{N}$:

```
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:

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

But with the Fun$N$-construction the equivalent type would be

```
Fun1<? extends Fun1<? extends Fun1<? extends C,
                                   ? super Fun2<? extends C,? super A,? super B>>,
                      ? super B>,
     ? 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 Fun$N$*, where the subtyping property is changed in comparison to Java. The special interfaces Fun$N$* correspond to functions types in Scala [Ode14].

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

```
interface FunN*<+R,-T1, ... , -TN>1 {
    R apply(T1 arg1, ... , TN argN);
}
```

where Fun$N$*<T_0,$T'_1, \ldots, T'_N$> $\leq^*$ Fun$N$*<T'_0,$T_1, \ldots, T_N$> iff $T_i \leq^* T'_i$ with $\leq^*$ as subtyping relation. For Fun$N*$ no wildcards are allowed.
Let us consider the following example

```
Object m(Integer x, Fun1*<Object, Integer> f) {
  return f.apply(x);
}
```

It is obvious, that the following application is correct:

```
Fun1*<Object,Integer> f_IntObj = ...
Object x2 = m(2, f_IntObj);
```

But for Integer $\leq^*$ Number $\leq^*$ Object also

```
Fun1*<Integer,Integer> f_IntInt = ...
Object x1 = m(2, f_IntInt);
```

is correct, as Fun1∗<Integer, Integer> is a subtype of Fun1∗<Object, Integer> and

```
Fun1*<Number, Number> f_NumNum = ...
Object x3 = m(2, f_NumNum);
```

is correct, as Fun1∗<Number, Number> is also a subtype of Fun1∗<Object, Integer>.

Example 2. Considering again Example 1 the program

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

has in Java-TX the type Fun1*<Fun1*<Fun1*<C,Fun2*<C,A,B»,B>,A>

1.2 Fun$N*$ as types of methods

We can also give Fun$N*$–types to methods. This means with the class CL

```
class CL {

  T0 meth (T1 x1, ... , TN xN) { ...  }

}
```

the method reference CL::meth has the type Fun$N*$<T_o,$T_1, \ldots, T_N$>.
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 `Fun`N`*`–type.
– A lambda expression fits any target type, which must be a functional interface, if its method's type in `Fun`N`*`–representation is a supertype of the explicit type.

Example 3. Let us consider again:

```
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 `Fun`N^* 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`.

$$
\begin{array}{l}
\mathbf{TI}\colon \texttt{TypeAssumptions} \times \texttt{Class} \to \{\, (\texttt{Constraints}, \texttt{TClass}) \,\} \\
\quad \mathbf{TI}(\, Ass, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau'\,), fdecls\,)\,) = \\
\quad\quad \mathbf{let}\ \underline{(\mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau'\,), fdecls_t\,), ConS)} = \\
\quad\quad\quad\quad \mathbf{TYPE}(\, Ass, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau'\,), fdecls\,)\,) \\
\quad\quad\quad\ \underline{\{\, (cs_1, \sigma_1), \ldots , (cs_n, \sigma_n) \,\}} = \mathbf{SOLVE}(\, ConS\,) \\
\quad\quad \mathbf{in}\ \{\, (cs_i, \sigma_i(\, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau'\,), fdecls_t\,)\,)) \mid 1 \leqslant i \leqslant n \,\}
\end{array}
$$

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 $(a \lessdot a')$ of types variables and σ_i consists of solutions $(a \doteq \theta)$, where $(a \lessdot a')$ means a has to be a subtype of a' and $(a \doteq b)$ 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>>`. `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 `op` is the application of the function (second argument) to its object (`this`) and its first argument. The method `mul` is the ordinary matrix multiplication in lambda representation. Finally, in `main` the function `op` is applied. The `op`-function of matrix `m1` is applied to the matrix `m2` and the function `mul` of `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 `ret; v1; v2; m1;` and `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 $\texttt{Fun}N^*$ types induces an extension of this unification. The three most important added unifications rule are given in Figure 3. In the rules $a \lessdot b$ means a must be a subtype of b and $a \doteq b$ 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

$$\text{(reduceFun}N^*) \quad \frac{Eq \cup \{\, \texttt{Fun}N\texttt{*<}\theta, \theta'_1, \ldots, \theta'_N\texttt{>} \lessdot \texttt{Fun}N\texttt{*<}\theta', \theta_1, \ldots, \theta_N\texttt{>} \,\}}{Eq \cup \{\, \theta \lessdot \theta', \theta_1 \lessdot \theta'_1, \ldots, \theta_N \lessdot \theta'_N \,\}}$$

$$\text{(greaterFun}N^*) \quad \frac{Eq \cup \{\, \texttt{Fun}N\texttt{*<}\theta, \theta'_1, \ldots, \theta'_N\texttt{>} \lessdot a \,\}}{Eq \cup \{\, a \doteq \texttt{Fun}N\texttt{*<}b', b_1, \ldots, b_N\texttt{>}, \theta \lessdot b', b_i \lessdot \theta'_i \,\}} \quad b', b_i \text{ are fresh}$$

$$\text{(smallerFun}N^*) \quad \frac{Eq \cup \{\, a \lessdot \texttt{Fun}N\texttt{*<}\theta', \theta_1, \ldots, \theta_N\texttt{>} \,\}}{Eq \cup \{\, a \doteq \texttt{Fun}N\texttt{*<}b, b'_1, \ldots, b'_N\texttt{>}, b \lessdot \theta', \theta_1 \lessdot b'_i \,\}} \quad b', b_i \text{ are fresh}$$

Fig. 3. Extension of the type unification

The rule **reduceFun*N***** describes the reduction of the FunN^* interfaces. This means that the parameters are in covariant respectively contravariant relations. The rules **greaterFun*N***** and **smallerFun*N***** describes the solutions of all greater respectively all smaller FunN^*-types. This means that the parameters of the FunN^*-types gets greater respectively smaller.

2.3 Example

In the following we show the functionality of the type inference algorithm **TI** by the application to the function `op` from Figure 2. First the function **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:

$$\begin{array}{l}
\texttt{op:}a_{op}\ \texttt{=} \\
\quad \underline{\texttt{((m:}\underline{a_m}\texttt{) ->}} \\
\qquad \texttt{((f:}\underline{a_f}\texttt{) ->f.apply(this:}\underline{\texttt{Matrix}}\texttt{, m:}\underline{a_m}\texttt{):}\underline{a_3}\texttt{)} \\
\qquad \texttt{:}\underline{\texttt{Fun1*<}a_{app}, a_f\texttt{>}}\texttt{)} \\
\quad \texttt{:}\underline{\texttt{Fun1*<}a_{\lambda f}, a_m\texttt{>}}
\end{array}$$

and the set of constraints is given as:

$$\begin{array}{l}
\{\, (\texttt{Fun1}{*}\texttt{<}a_{\lambda f}, a_m\texttt{>} \lessdot a_{\texttt{op}}), (\texttt{Fun1}{*}\texttt{<}a_{app}, a_f\texttt{>} \lessdot a_{\lambda f}), \\
\;\; (a_f \doteq \texttt{Fun2}{*}\texttt{<}a_3, a_1, a_2\texttt{>}), (\texttt{Matrix} \lessdot a_1), (a_m \lessdot a_2), \\
\;\; (a_3 \lessdot a_{app}) \,\}
\end{array}$$

With applying **greaterFun*N***** to $\texttt{Fun1}{*}\texttt{<}a_{\lambda f}, a_m\texttt{>} \lessdot a_{\texttt{op}})$ we get

$$\{\, (a_{\texttt{op}} \doteq \texttt{Fun1}{*}\texttt{<}b', b_1\texttt{>}), (a_{\lambda f} \lessdot b'), (b_1 \lessdot a_m) \,\}.$$

With applying **greaterFun*N***** to $\texttt{Fun1}{*}\texttt{<}a_{app}, a_f\texttt{>} \lessdot a_{\lambda f}$ we get

$$\{\, (a_{\lambda f} \doteq \texttt{Fun1}{*}\texttt{<}c', c_1\texttt{>}), (a_{app} \lessdot c'), (c_1 \lessdot a_f) \,\}.$$

With substituting $a_{\lambda f}$ in $a_{\lambda f} \lessdot b'$ and again applying **greaterFun*N***** we get

$$\{\, (b' \doteq \texttt{Fun1}{*}\texttt{<}d', d_1\texttt{>}), (c' \lessdot d'), (d_1 \lessdot c_1) \,\}$$

With substituting a_f in $c_1 \lessdot a_f$ and applying **smallerFun*N**** we get

$$\{\, (c_1 \doteq \texttt{Fun2*<}x, x'_1, x'_2\texttt{>}), (x \lessdot a_3), (a_1 \lessdot x'_1), (a_2 \lessdot x'_2) \,\}$$

With substituting c_1 in $d_1 \lessdot c_1$ and applying **smallerFun*N**** again we get

$$\{\, (d_1 \doteq \texttt{Fun2*<}y, y'_1, y'_2\texttt{>}), (y \lessdot x), (x'_1 \lessdot y'_1), (x'_2 \lessdot y'_2) \,\}$$

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

$$\begin{array}{l}
\{\, \texttt{Matrix} \lessdot a_1 \lessdot x'_1 \lessdot y'_1 \\
\;\; b_1 \lessdot a_m \lessdot a_2 \lessdot x'_2 \lessdot y'_2, \\
\;\; y \lessdot x \lessdot a_3 \lessdot a_{app} \lessdot c' \lessdot d', \\
\;\; a_{\texttt{op}} \doteq \texttt{Fun1*<Fun1*<}d', \texttt{Fun2*<}y, y'_1, y'_2\texttt{>>}, b_1\texttt{>}, \\
\;\; a_{\lambda f} \doteq \texttt{Fun1*<}c', \texttt{Fun2*<}x, x'_1, x'_2\texttt{>>}, \\
\;\; a_f \doteq \texttt{Fun}\mathit{2}\texttt{*<}a_3, a_1, a_2\texttt{>} \,\}
\end{array}$$

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

$$\begin{array}{l}
\{\, (\{\, \texttt{b}_1 \lessdot a_2 \lessdot x'_2 \lessdot y'_2, \\
\qquad y \lessdot x \lessdot a_3 \lessdot c' \lessdot d' \,\}, \\
\quad \{\, a_{\texttt{op}} \doteq \texttt{Fun1*<Fun1*<}d', \texttt{Fun2*<}y, y'_1, y'_2\texttt{>>}, b_1\texttt{>}, \\
\qquad a_{\lambda f} \doteq \texttt{Fun1*<}c', \texttt{Fun2*<}x, x'_1, x'_2\texttt{>>}, \\
\qquad a_f \doteq \texttt{Fun}\mathit{2}\texttt{*<}a_3, a_1, a_2\texttt{>} \,\}) \\
\;\mid\;\; \texttt{Matrix} \leq^* a_1 \leq^* x'_1 \leq^* y'_1 \,\}
\end{array}$$

The result of **TI** is given as the application of the **SOLVE**'s results to the result program of **TYPE**. The result consists of a set of typings for `op`:

```
class Matrix extends Vector<Vector<Integer>> {

    <y2’, b1 extends y2’, d’, y extends d’>[2]
    Fun1*<Fun1*<d’, Fun2*<y,  X , y2’>>, b1>
        op = (m) -> (f) -> f.apply(this, m);
    ...
}
```

where `Matrix` $\leq^* X$.

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 (`Matrix` $\leq^* X$).

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.

```
class OL {

    m(a) { return a + a; }
    m(a) { return a || a; }
}


class Main {

    main(a) {
        ol;
        ol = new OL();
        return ol.m(a);
    }
}
```

Fig. 4. Type inference in the presence of overloading

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 `main(`X `a) {`
      `OL ol;`
      `ol = new OL();`
      `return ol.m(a); }`$|$ $X \in \{$ `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 `Fun`N`*`–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

[2] The constraints are given here as bounded type variables for fields, which is permitted only in methods in Java

```
op : Fun1*<Fun1*<d',Fun2*<y,Vector<? extends Vector<? extends Integer>>, y2'>>,b1>
   & Fun1*<Fun1*<d',Fun2*<y,Vector<? super Vector<? super Integer>>, y2'>>,b1>
   & ... &
   & Fun1*<Fun1*<d',Fun2*<y,Vector<Vector<Integer>>, y2'>>,b1>
   & Fun1*<Fun1*<d',Fun2*<y,Matrix, y2'>>,b1>
```

Fig. 5. Intersection type of `op`

```
op : Fun1*<Fun1*<d',Fun2*<y,Vector<? extends Vector<? extends Integer>>, y2'>>,b1>
   & Fun1*<Fun1*<d',Fun2*<y,Vector<? extends Vector<? super Integer>>, y2'>>,b1>
   & Fun1*<Fun1*<d',Fun2*<y,Vector<? super Vector<Integer>>, y2'>>,b1>
```

Fig. 6. Reduced intersection type of `op`

between elements of the intersection. Therefore a better approach would be to define the type of `op` as the intersection of all maximal elements in the subtyping ordering. Then the type of `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 `id: a -> a`, where `a` is a type variable. This means all other types of `id` are instantiations of `a -> a`, e.g. `id: int -> int` or `id: char -> 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 `Matrix<E>` and an additional class `intMatrix`, that contains the method `mul` (cp. Figure 7).
The type of `op` applied in the method `main` is

```
Fun1*<Fun1*<intMatrix, Fun2*<intMatrix,intMatrix, intMatrix>>, intMatrix>.
```

The corresponding element of the principal intersection type is

```
Fun1*<Fun1*<d', Fun2*<y,Vector<? extends Vector<? extends E>>, y2'>>, b1>
```

```
class Matrix<E> extends Vector<Vector<E>> {

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

class IntMatrix extends Matrix<Integer>

    mul = (m1, m2) -> { ... }


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

Fig. 7. Parametrized Matrix

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

$$\texttt{main} : \texttt{Integer} \to \texttt{Integer} \,\&\, \texttt{String} \to \texttt{String} \,\&\, \texttt{Long} \to \texttt{Long} \,\&\, \texttt{Double} \to \texttt{Double} \,\&\, \texttt{Boolean} \to \texttt{Boolean} \,\&\, \texttt{Float} \to \texttt{Float}$$

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

```
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);
    }
}
```

Fig. 8. The class `Put`

The principal type of `main` is:

$$\texttt{main} : \texttt{T} \times \texttt{Vector<T>} \to \texttt{void} \,\&\, \texttt{T} \times \texttt{Stack<T>} \to \texttt{void}.$$

If the call-graph would not be considered, $\texttt{T} \times \texttt{Stack<T>} \rightarrow \texttt{void}$ would not belong to the principal type, as `Stack` is a subtype of `Vector`. But this type is necessary as `main` defines different functions on `Vector` and `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 `OL` and `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

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.

[GJS+14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Language Specification*. The Java series. Addison-Wesley, Java SE 8 edition, 2014.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.

[Ode14] Martin Odersky. *The Scala Language Specification Version 2.9*, May 2014.

[ORW00] Martin Odersky, Enno Runne, and Philip Wadler. Two Ways to Bake Your Pizza – Translating Parameterised Types into Java. *Proceedings of a Dagstuhl Seminar, Springer Lecture Notes in Computer Science*, 1766:114–132, 2000.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with Wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

[Plü08] Martin Plümicke. Intersection Types in Java. In Luís Veiga, Vasco Amaral, Nigel Horspool, and Giacomo Cabri, editors, *6th International Conference on Principles and Practices of Programming in Java*, volume 347 of *ACM International Conference Proceeding Series*, pages 181–188, September 2008.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

[Plü11] Martin Plümicke. Well-typings for Java$_\lambda$. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 91–100, New York, NY, USA, 2011. ACM.

[Plü14] Martin Plümicke. Functional Interfaces vs. Function Types in Java with Lambdas – Extended Abstract. In *Tagungsband der Arbeitstagung Programmiersprachen (ATPS 2014)*, volume Vol-1129, pages 146–147. CEUR Workshop Proceedings (CEUR-WS.org), 2014.

[Plü15] Martin Plümicke. More type inference in java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 252–265, 1998.

[Sta15] Andreas Stadelmeier. Java type inference as an Eclipse plugin. In *Proceedings of the Studierendenkonferenz Informatik SKILL 2015*, 2015. (to appear).

[vB93] Steffen van Bakel. Principal type schemes for the strict type assignment system. *Journal of Logic and Computing*, 3(6):643–670, 1993.