Introducing Scala-like function types into Java-TX

Martin Plümicke
Cooperative State University Baden–Württemberg
Horb, Baden–Württemberg
pl@dhbw.de

Andreas Stadelmeier
Cooperative State University Baden–Württemberg
Horb, Baden–Württemberg
a.stadelmeier@hb.dhbw-stuttgart.de

ABSTRACT

This paper considers the realisation of lambda expressions in Java 8 on the basis of a global type inference algorithm, which we have introduced in Java-TX. We demonstrate that the Java 8 approach has indeed some benefits but also a number of drawbacks. In order to eliminate the drawbacks, we take into consideration the approaches in a former experimental Java version (strawman approach) and in Scala. We show that an integration of these approaches eliminates the drawbacks without losing the benefits of the Java 8 approach. Additionally, we adapt our global type inference algorithm to this extended language.

CCS CONCEPTS

• **Software and its engineering → Object oriented languages**; **Language features**; *Compilers*;

KEYWORDS

Java, lambda expressions, function types, type inference

ACM Reference Format:
Martin Plümicke and Andreas Stadelmeier. 2017. Introducing Scala-like function types into Java-TX. In *Proceedings of ManLang 2017, Prague, Czech Republic, September 27–29, 2017,* 12 pages.
https://doi.org/10.1145/3132190.3132203

1 INTRODUCTION

Java-TX (Java Type eXtended) is an extension of Java 8, in which a global type inference algorithm is added. Since the end of the nineties features from functional programming languages have been transferred to object-oriented languages. One of the first approaches where PIZZA [9]. Three ideas had been incorporated into PIZZA: parametric polymorphism, higher-order functions, and algebraic data types. The parametric polymorphism extended by wildcards, called generics, were then transfered to Java 5.0 [5]. Higher-order functions and lambda expression were introduced in Java 8 [6]. Java 8 uses functional interfaces as target types of lambda expressions in contrast to real function types used by PIZZA. In Scala [17], C# [2], and C++ [20] generics and higher-order functions are also introduced.

ManLang 2017, September 27–29, 2017, Prague, Czech Republic
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5340-3/17/09…$15.00
https://doi.org/10.1145/3132190.3132203

The powerful feature *type inference* from functional programming languages is incorporated into object-oriented languages only in a restricted way called local type inference [10, 11]. Local type inference allows certain type annotations to be omitted. It is, for instance, often not necessary to specify the type of a variable. Type parameters of classes in the new–statement can be left out. Return types of methods can often also be omitted. Local type inference is at its most pronounced in Scala. In contrast to global type inference, local type inference allows types of recursive methods and lambda expressions not to be omitted.
The Java-TX project contributes to the design of object-oriented languages by developing global type inference algorithms for Java-like languages. In [12] we presented a type inference algorithm for a core Java 5.0 including wildcards. Further on we gave type inference algorithms for the language of the strawman approach [14] and for a core of Java 8 [16].
In this paper we will consider our Java-TX type inference algorithm more in detail. We will recognise it infers indeed correct but sometimes less reasonable types. Furthermore, we will show, this is not caused by the algoritm itself. This is due to a number of drawbacks to the implementation of lambda expressions in Java 8.
Therefore, we consider the types of lambda expressions used by Java 8 in detail. We carve out the benefits and the drawbacks. We discuss different alternatives and offer a solution to the problems. Finally, we present a changed type inference algorithm for the changed language Java-TX.
The paper is structured as follows: In the next section we consider lambda expressions in Java 8 in detail. We present the benefits and show that direct application of lambda expressions and subtyping of function types have some disadvantages. In the third section we go on to present alternative approaches. We consider lambda expressions in the strawman approach and in Scala. In the forth section we present our type inference algorithm for the language of the strawman approach and for a core of Java 8. In the fifth section we discuss the different approaches and justify the decision to introduce Scala-like function types into Java-TX. In the sixth section we give a specification for this integration, involving an alteration of the original Java 8 specification. In Section seven and eight we present the adaptions of the type inference and the type unification algorithm caused by the change of the language respectively. We end with a conclusion and an outlook to future work.

2 LAMBDA EXPRESSIONS IN JAVA 8

With Java 8 we saw the introduction of Lambda expressions. After lengthy discussion the decision was taken to introduce functional interfaces as target types of lambda expression instead of real function types. There are a number of substantial arguments against real function types, as expressed in [4]:

- It would add complexity to the type system and further mix structural and nominal types (Java is almost entirely nominally typed).
- It would lead to a divergence of library styles – some libraries would continue to use callback interfaces, while others would use structural function types.
- The syntax could be unwieldy, especially when checked exceptions were included.
- It is unlikely that there would be a runtime representation for each distinct function type, meaning developers would be further exposed to and limited by erasure. For example, it would not be possible (perhaps surprisingly) to overload methods `m(T->U)` and `m(X->Y)`.

In this section we will first give a short overview of lambda expressions in Java 8 and we will show the benefit of using functional interfaces as target types of lambda expressions. Subsequently we will consider various difficulties the decision may pose. Furthermore, on the one hand we will consider direct applications of lambda expressions and on the other hand we will consider the subtyping property.
In this paper we denote types, representing classes and interfaces, as *class types*, in contrast to which there are *function types* (e.g. $(\theta_1, \ldots, \theta_n) \to \theta_0$) and *base types* (e.g. `int`, `boolean`, `char`, etc.)

2.1 Functional interfaces as target types of lambda expressions

The Java language specification [6] defines the language, its syntax and semantics. The Lambda expressions are defined in §25.27, whereas the types of lambda expressions are defined in §15.27.3. In the following we will cite the two most important definitions for *functional interfaces* (defined in §9.8) and *compatible target types* (defined in §15.27.3). For the other definitions we refer to the language specification [6].

Definition 2.1 (Functional interface). For an interface `I`, let M be its set of abstract methods that do not have the same signature as any public method of the class `Object`. Then, `I` is a *functional interface* if there exists a method `m` in M for which both of the following are true:

- The signature of `m` is a subsignature of every method's signature in M.
- m is return-type-substitutable (§8.4.5 [6]) for every method in M.

Example 2.2. An simple function interface is an interface with one abstract method that does not have the same signature as any public instance method of the class `Object`.

```
interface I {
  int compare(String o1, String o2);
}
```

A more complex example is presented here:

Example 2.3. Let the interfaces `X`, `Y`, and `Z` be given.

```
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<String> arg); }
interface Z extends X, Y {}
```

In addition to the interfaces `X` and `Y`, the interface `Z` is also a functional interface. As both methods have the same signature, the inherited methods logically represent a single method.

Definition 2.4 (Lambda expressions compatible with a target type). A lambda expression is *compatible* in an assignment context, invocation context, or casting context with a *target type* T if T is a functional interface type and the expression is congruent with the function type of the ground target type derived from T, which means

- The function type has no type parameters.
- The number of lambda parameters is the same as the number of parameter types of the function type.
- If the lambda expression is explicitly typed, its formal parameter types are the same as the parameter types of the function type.
- If the lambda parameters are assumed to have the same types as the function type's parameter types, then:
  - If the result of the function type is `void`, the lambda body is either a statement expression (§14.8 [6]) or a `void`-compatible block.
  - If the result of the function type is a (non-`void`) type R, then either
    - i the lambda body is an expression that is compatible with R in an assignment context, or
    - ii the lambda body is a value-compatible block, and each result expression (§15.27.2 [6]) is compatible with R in an assignment context.

For the exact definitions of *function type of the ground target type* we refer to [6]. In Section 6 we will extend this definition.
We will give an example that illustrates these definitions.

Example 2.5. Given the functional interfaces `Function` and `BiFunction` from the package `java.util.function`:

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

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

First of all, let the declaration

```
  BiFunction<Integer, Integer, Integer>
  f = (Integer x, Integer y) -> x * y;
```

be given. The ground target type is `BiFunction<Integer, Integer, Integer>`. Its function type is

```
(Integer, Integer) -> Integer.
```

The lambda expression

```
(Integer x, Integer y) -> x * y;
```

is congruent with the function type as the function type as well as the lambda expression have two parameters of the type `Integer`. Furthermore, the function type's result type is `Integer` and under the assumptions that `x`'s and `y`'s type are `Integer`, the expression `x * y` is compatible with `Integer`.

And secondly, let us consider a more complex example:

Example 2.6. Let the typed lambda expression f be given.

```
Function<? super Integer,
          Function<? super Integer,
                    ? extends Integer>>
f = x -> y -> x * y;
```

The ground target type is

```
Function<Integer,
         Function<? super Integer, ? extends Integer>>.
```

Its function type is

```
(Integer) ->
           Function<? super Integer, ? extends Integer>.
```

For the congruence of the lambda expression with the function type, we have to show that the function type and the lambda expression have the same number of arguments. This is obvious. Furthermore we have to show that if the type of x is assumed as Integer, then the lambda expression y -> x * y is compatible with the function's type result type Function<? super Integer, ? extends Integer>. The ground target type of this type is Function<Integer, Integer>. Its function type is (Integer) -> Integer. The number of arguments are equal. Finally we have to consider the lambda body. This must be compatible with the function type's result type. Under the additional assumption that y's type is Integer, too, x * y is compatible with Integer.

The major benefit of introducing lambda expressions with compatible target types is the possibility of implementing callback functions in existing libraries by means of lambda expressions. In general, callback functions are function parameters of libraries. In programming languages such as C they are realised by function pointers. Until version 7 in Java there had been nothing like function pointers, such that callback functions are implemented by interfaces with one abstract method. This has often been implemented by anonymous inner classes. The following JavaFX example, borrowed from Oracle's get-started-tutorial, illustrates the situation:

```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.control.Button;
...
Button btn = new Button();
btn.setText("Say␣'Hello␣World'");
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

The anonymous inner class, underlined in the code, implements the event-handler. In Java 8 this callback function is implemented by a lambda expression.

```
...
btn.setOnAction(
    event -> System.out.println("Hello World!")
);
```

In summary then, introducing lambda expressions with compatible target types allows callback function to be implemented by lambda expressions without having to change the millions of Java libraries in existence worldwide.

However, despite this benefit, there are some disadvantages of the target typing. In the following sections we will present two problems, namely, both the direct applications of lambda expressions and the subtyping of function types.

2.2 Direct application of lambda expressions

In the λ-calculus (e.g. [1]) β-conversion (direct application of a lambda expression to its arguments) the following 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. Lambda expressions are only permitted in an assignment, invocation, or casting context (cp. Def. 2.4), the reason being that lambda expressions themselves have no types in Java 8. As there is no target type given, the corresponding functional interface is unknown so that the name of the method is unknown, too, or else the method apply could not be identified. A target type can be introduced by a type-cast. Therefore the functional interface Function from the package java.util.function has to be used again:

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

Now the interface Function and the method apply are both known. Generally speaking for the application of a function in the curried representation

$$f : T_1 \to \ldots \to T_N \to T_0,$$

as type-cast a nested functional interface is needed:

```
((Function<T1, Function<T2, ... Function<TN, T0>>> )
 x1 -> x2 ->. . . -> xN -> h(x1, . . . ,XN))
 .apply(a1).apply(a2). . . . .apply(aN)
```

Summarising then, the absence of lambda expressions' types necessitates type-casts for using direct lambda expression applications. Indeed, the type-cast presents a possibility for implementing direct lambda expression applications, but it is unhandy.

2.3 Subtyping of function types

Another important object-oriented feature is subtyping. Therefore subtyping should be given for function types, too. We denote the subtyping relation by $\leq^*$.
In Java 8 subtyping can also be implemented by the functional interfaces from the package java.util.function.
Mathematical functions have the property that the argument types are contravariant and the return types are covariant. This means for a subtype

$$(arg\_type' \longrightarrow ret\_type) \leq^* (arg\_type \longrightarrow ret\_type')$$

the following is true:

$$arg\_type \leq^* arg\_type' \text{ and } ret\_type \leq^* ret\_type'.$$

As Java 8 allows no declaration-site variance, this property has to be expressed by wildcards:

Function<arg_type', res_type> $\leq^*$
    Function<? super arg_type, ? extends res_type'>

This is a possibility for implementing subtyping of function types, but it is counterintuitive. Let us consider an example. The class

```
import java.util.function.*;

class MathStruc<A> {
  A model;

  Function<BiFunction<A,A,A>,
           Function<MathStruc<A>,MathStruc<A>>>
  innerOp = (o) -> (ms) ->
    new MathStruc<>(o.apply(this.model,ms.model));

  MathStruc(A m) { model =m; }
}
```

Figure 1: Class MathStruc in Java 8

MathStruc (Fig. 1) represents mathematical structures such as vector spaces, monoids, groups, etc. with an inner operation o. The concrete model of the mathematical structure is given in the field model. The inner operation o takes two arguments of the model's type and results in the model's type. The field innerOp is the function that applies the operation o to two mathematical structures. The function is given in a so-called curried representation. The implementation is abstracted from a concrete operation o, which is the first argument of the lambda expression. The lambda expression results in another lambda expression that takes the second argument of o and results in the application of the operation o to the model of this and the model of the second argument. In Fig. 2 an application for the structure (Integer, +) is given.

```
MathStruc<Integer> msi = new MathStruc<>(1);
MathStruc<Integer> res =
  msi.innerOp.apply((i, j) -> i + j)
  .apply(new MathStruc<>(3));
```

Figure 2: Application for the structure (Integer, +)

If we want to enable subtypes for the function types, we have to change the type declaration of innerOp as given in Fig. 3.

```
  Function<? super BiFunction<? super A,
                              ? super A,
                              ? extends A>,
           ? ext. Function<? super MathStruc<A>,
                           ? ext. MathStruc<A>>>
  innerOp = ...
```

Figure 3: Enabling subtypes in innerOp

Besides the illegible syntax, the application in Fig. 2 is no any longer correct. As ? super BiFunction<...> is not a compatible target type for (i, j) -> i + j, similar to hat we saw in Section 2.2, a type-cast has to be introduced (Fig. 4). This is again counterintuitive.

```
MathStruc<Integer> msi = new MathStruc<>(1);
MathStruc<Integer> res =
  msi.innerOp
  .apply((BiFunction<Integer,Integer,Integer>)
         (i, j) -> i + j)
  .apply(new MathStruc<>(3));
```

Figure 4: Inserted type-casts

In this special case introducing a wildcard can be avoided. As in the program in Fig. 1 only a subtype of o's type BiFunction<? super A, ? super A, ? extends A> is used, and no subtype of innerOp's type, the wildcard can be omitted so that the type of innerOp can be given as in Fig. 5. Then, no additional type-cast is necessary as was the case in Fig. 4.

```
Function<BiFunction<? super A,
                    ? super A,
                    ? ext. A>,
         ? ext. Function<? super MathStruc<A>,
                         ? ext. MathStruc<A>>>
innerOp = ...
```

Figure 5: Reduced wildcards

In Java only use-site variance of type parameters is allowed. We will now proceed to show that use-site variance, when applied to the interface Function, leads to some less reasonable but correct declarations.
Let us consider the example in Fig. 6. There are functions f, g1, and g2. f are typed by Function<Sub, Sub> and g1 by Function<? super Sub, ? extends Super>. As explained above, f's type is a subtype of g1's, such that the assignment g1 = f is correct and the application g1.apply(new Sub()) resulting in the type Super is correct, too.
Otherwise, following Java's use-site variance, it is possible to type g2 by Function<? extends Super, ? super Sub>. Function<Sub, Sub> is also a subtype of Function<? extends Super, ? super Sub>. Therefore assigning f to g2 is type-correct, too. But g2 can not be applied to objects of any declared type, especially not to any Sub. g2 can only be applied to null. This is in Java possible as all methods are applicable to null. The result of this application can only be of the type Object. This is in Java possible as every method's result type is Object.
To summarize: Subtyping of function types can be simulated in Java 8 by functional interfaces from the package java.util.function. The contravariance of argument types and the covariance of the result types have to be simulated by the super and the extends wildcard, respectively. General, additional type-casts have to be introduced for higher order functions (functions as arguments or results of other functions), as wildcard-types are not target-types of lambda expressions. Following the use-site variance of type parameters, it possible to declare less reasonable subtyping relations of

```
class Super { }

class Sub extends Super { }

class Main {
    void m() {
      Function<Sub, Sub> f = x -> x;

      //common subtyping
      Function<? super Sub, ? extends Super> g1;
      g1 = f;
      Super sup = g1.apply(new Sub());

      //less reasonable subtyping
      Function<? extends Super, ? super Sub> g2;
      g2 = f;
      //g2.apply(new Sub()); //incorrect
      g2.apply(null);
      //sup = g2.apply(null); //incorrect
      Object o = g2.apply(null);
    }
}
```

Figure 6: Less reasonable subtyping in Java 8

function types. The approach to model function types by the interfaces of the package `java.util.function` is not quite beautiful, but it works.
In this paper we will offer a proposal of how to preserve the benefits of the target typing and to solve the problems presented here.

3 DIFFERENT APPROACHES TO LAMBDA EXPRESSIONS IN JAVA-LIKE LANGUAGES

In this section we will take a look on two different approaches to lambda expressions. First we will consider the strawman approach, that was an approach which dates back to the early phase of Project Lambda[1], but it was not realised. After that we consider the approach of lambda expressions in Scala.

3.1 Strawman approach

In early stage of the Project Lambda undertaken by the Sun Microsystems, Inc. [7] plan was to introduce lambda expression and real function types into Java. Furthermore, the idea that lambda expressions could implement an interface with only one method was also considered. We call this language in the following Java$_\lambda$.
The syntax of the lambda expressions in Java$_\lambda$ was given by

$$\#\,(t_1\ x_1, \ldots, t_n\ x_n) \rightarrow (\ \textit{expression} \mid \textit{block}\ ).$$

The syntax of the corresponding function types was given by

$$\#t_o(t_1, \ldots, t_n).$$

The syntax of the function type is given in the common Java-style, which means, the result type is given on the left side of the argument types. The # symbol is also employed.

[1]http://openjdk.java.net/projects/lambda

```
class MathStruc<A> {

  A model;

  ##MathStruc<A>(MathStruc<A>)(#A(A,A))
  innerOp = #(#A(A,A) o)->#(MathStruc<A> ms)->
    new MathStruc<A>(o.(this.model,ms.model));
}
```

Figure 7: Class `MathStruc` in the strawman approach

Furthermore, the idea of implementing functional interfaces (in [7] called SAM–types) by lambda expression is already given. This means the strawman approach contains real function type as well as functional interface as target types of lambda expressions.
The subtyping relation is given in the usual way:

$$\#t_o(t'_1, \ldots, t'_n) \leq^* \#t'_o(t_1, \ldots, t_n),$$

iff $t_i \leq^* t'_i$, for $0 \leqslant i \leqslant n$.
The `MathStruc` example written in the language of strawman approach language Java$_\lambda$ (Fig. 7) serves to illustrate some problems in the design of the strawman's language Java$_\lambda$.
There are two problems in particular: On the one hand the double use of # is confusing and on the other hand the Java-style writing of function types is often difficult to read. It is not obvious that the type of `innerOp`

```
##MathStruc<A>(MathStruc<A>) (#A(A,A))
```

means the function type

```
((A,A)->A) -> (MathStruc<A> -> MathStruc<A>).
```

Furthermore, the use of "." as apply-constructor is another *overloading* of the dot operator. This makes no contribution to the clarity of the language. Additionally, the language definition demands, that all arguments of lambda expressions are explicitly typed. This means a lambda expression such as `#(o) -> #(ms) -> ...` is not allowed.
On the other hand the real function types allow the direct application lambda expressions. No additional type-cast is necessary. Furthermore there is simple approach to the subtyping of given function types, whereby wildcard-arguments are not necessary.
In summary, the strawman approach allows real function types alongside the possibility of implementing functional interfaces by means of lambda expression. This integration is an advantage in comparison to the loss of real function types in Java 8.
A big disadvantage is the reduced readability of the programs, as the double use of # and the Java-style writing of function types. A further disadvantage is that there is a mix of structural and nominal types.

3.2 Scala

Scala contains lambda expressions (anonymous functions) and real function types.
Anonymous functions are declared by

$$(\mathtt{x}_1[\mathtt{:\ t}_1], \ldots, \mathtt{x}_n[\mathtt{:\ t}_n])\ \mathtt{=>}\ (\ \textit{expression} \mid \textit{block}\ ),$$

where the types of the arguments can be omitted if they are defined in the expected type.
Scala allows the variance of class parameters to be declared. This means it could be declared whether a type parameter is covariant or contravariant. For example, the covariant declaration

```
class C[+A] { ... }
```

means that `C[T1]<:C[T2]`[2] iff `T1<:T2` and the contravariant declaration

```
class C[-A] { ... }
```

means that `C[T2]<:C[T1]` iff `T1<:T2`.
In Scala there are two variants of the syntax of function types.

```
(T1, ..., Tn) => U
```

is a shorthand for the class type

```
Functionn[-T1, ...,-Tn,+U],
```

where `T1` , . . . , `Tn` are contravariant and `U` is covariant. This means that the subtyping relation of function types in Scala are defined as expected.
Similar to Java 8's functional interface `Function` the class types `Functionn[T1, ..., Tn,U]` contain the method `apply`.
In Scala, in general, function applications $f(expr_1, \dots, expr_n)$ are possible. If f has a function type, the application is equivalent to f.apply$(expr_1, \dots, expr_n)$.
Let us consider the class `MathStruc` in Scala (Fig. 8).

```
class MathStruc[A](var model : A) {

  var innerOp:Function1[Function2[A,A,A],
                        Function1[MathStruc[A],
                                  MathStruc[A]]] =
  f => ms =>
       new MathStruc[A](f(this.model,ms.model));}
```

Figure 8: Class `MathStruc` in Scala

In Scala both problems of lambda expressions in Java 8 that we have highlighted are solved. Direct applications of lambda expressions need no type-casts and subtyping of function types is allowed in an normal way as the argument types are contravariant and the result type is covariant.
In contrast to Java 8, in Scala lambda expressions can not be used to implement functional interfaces.

Local type inference. In Scala local type inference [10, 11] is implemented. Often it is not necessary to specify the type of a variable. The type checker deduces the type from the expression. In the above example the type can be omitted:

```
var innerOp = f => ms =>
  new MathStruc[A](f(this.model,ms.model));
```

For recursive definitions of lambda expression (cp. letrec in [3]) the type cannot be inferred.
The following example does not compile.

[2]In Scala `T1<:T2` means `T1` is a subtype of `T2`.

```
val fact = (x) => if (x <= 1) 1
                  else x * fact(x-1);
```

It is necessary to give an explicit type annotation.

```
val fact:Function1[Integer,Integer] =
  (x) => if (x <= 1) 1
         else x * fact(x-1);
```

The main feature of Java-TX is *global* type inference. This means no type annotation is necessary. Especially recursive programs like `fact` do not need any type annotations. We will present this example with Java-TX global type inference at the end of Section 8.

4 OUR TYPE INFERENCE ALGORITHMS

For a core of the languages Java$_\lambda$ of the strawman approach and of Java 8 we gave *global* type inference algorithms [14–16, 19]. The principle of the algorithms is that first a set of constraints (subtyping-inequations) are determined. Then these constraints are solved by a type unification algorithm [13]. As the type unification problem is not unitary but finitary, there are, in general, finitely many results. To deal with the different results, we implemented an Eclipse plugin [18].
In the next two sections we will consider the two algorithms in more detail and offer an example to illustrate the algorithm.

4.1 Type inference algorithm for the strawman approach

The algorithm [14] improved in [15] has three parts:

TYPE: The function **TYPE** introduces type annotations (fresh type variables or known types) to each subterm of the input expressions and determines the constraints.
MATCH: The function **MATCH** determines all substitutes which are function types and reduces all constraints to atomic constraints which consist only of class types or type variables.
TUnify: The function **TUnify** unifies the atomic constraints. The result is a set of unifiers and remaining contraints which consist only of type variables.
In the algorithm we denote $\theta \lessdot \theta'$ for two type terms, which should be type unified such that $\sigma(\theta) \leq^* \sigma(\theta')$ for a most general unifier σ. During the unification algorithm $\lessdot$ is replaced by $\lessdot_?$ and $\doteq$, respectively. $\theta \lessdot_? \theta'$ means that the two parameter types θ and θ' of type terms should be unified such that $X\!<\dots,\sigma(\theta),\dots\!>$ is a subtype of $X\!<\dots,\sigma(\theta'),\dots\!>$ and $\doteq$ is the usual unification.

Let us now consider again the Java program in Fig. 6 from Section 2.3. Omitting all type annotations, first we determine the principal type by our type inference algorithm. After that we instantiate some type variable to get the analogous situation as in Section 2.3. The shortened untyped program in Java$_\lambda$ is given as:

```
void m() {
  f = #(x) -> x;
  var g;³
  g = f;
}
```

Now we apply the type inference to this program. The result of the function **TYPE** is given by the type annotated program

```
void m() {
  f:α = (#(x:β) -> x:γ):δ;
  var g:ϵ;
  g:ϵ = f:α;
}
```

and the set of constraints: $\{\, \#\,\gamma\,(\beta) \lessdot \delta, \delta \lessdot \alpha, \alpha \lessdot \epsilon \,\}$.
The function **MATCH** determines the function type substitutes:

$$\delta \mapsto \#\,\gamma_1\,(\beta_1), \alpha \mapsto \#\,\gamma_2\,(\beta_2), \epsilon \mapsto \#\,\gamma_3\,(\beta_3)$$

and the set of atomic constraints:

$$\{\, \gamma \lessdot \gamma_1, \beta_1 \lessdot \beta, \gamma_1 \lessdot \gamma_2, \beta_2 \lessdot \beta_1, \gamma_2 \lessdot \gamma_3, \beta_3 \lessdot \beta_2 \,\}.$$

The function **TUnify** does nothing as all atomic constraints are in solved form. This means the types of f and g are given as:

$$\begin{aligned} &\mathtt{f} : \#\,\gamma_2\,(\beta_2),\\ &\mathtt{g} : \#\,\gamma_3\,(\beta_3) \end{aligned}$$

and it holds that $\#\,\gamma_2\,(\beta_2)$ is a subtype of $\#\,\gamma_3\,(\beta_3)$, as $\gamma_2 \leq^* \gamma_3$ and $\beta_3 \leq^* \beta_2$.
As in the example in Section 2.3 we annote f by the type # Sub (Sub). Then for the result type of g (namely γ_3) we can instantiate Sub as well as Super. But for the argument type of g (namely β_3) only Sub is allowed to be instantiated.
This means our type inference algorithm for the language of the strawman approach Java$_\lambda$ provides the desired result.
In the section that follows we will go on to consider the type inference algorithm for the approach with function interfaces.

4.2 Functional interfaces

In [16, 19] we presented a type inference algorithm for a core of Java 8. The core language in [16] supports mostly all features of Java 8 including generics and lambda expressions. Only methods are simulated by lambda expressions of fields. This leads to the restriction that overloading is impossible. In [19] methods including overloading are added. In both approaches we extend Java 8 by functional interfaces

```
R FunN<A1, ... AN, R>⁴{
  R apply(A1 arg1, ... AN);
}
```

analogous to the traits Function*N* in Scala. But in Java 8 the parameters of the functional interfaces have use-site variance, while in Scala the parameters have declaration-site variance. The type inference algorithm determines these types for lambda expressions.

[3] In Java-TX a local variable has to be declared before use. This is done by the keyword var.
[4] In [16] we gave the result type parameter as first parameter of Fun*N*. In this presentation we moved it to the last parameter so that notation is more similar to the interfaces for java.util.function.

The type inference algorithm **TI** consists of two functions **TYPE** and **SOLVE**. The function **TYPE** has had one change made to it compared to the above algorithm: The function types Fun*N* are already introduced into the code.
The function **SOLVE** unifies the constraints. A function **MATCH** is no any longer necessary as the type Fun*N* are class types, too.
In Section 7 we give the functions **TI** in Fig. 10, **TYPE** in Fig. 11, and **SOLVE** in Fig. 13, explicitly.
Let us consider the same example.

```
void m() {
  f = (x) -> x;
  var g;
  g = f;
}
```

The result of the function **TYPE** is given by the analogous type annotated program

```
void m() {
  f:α = ((x:β) -> x:γ):Fun1<β, γ>;
  var g:ϵ;
  g:ϵ = f:α;
}
```

and the set of constraints:

$$\{\, \mathtt{Fun1<}\beta,\ \gamma\mathtt{>} \lessdot \alpha, \alpha \lessdot \epsilon \,\}.$$

The result, pairs of unifiers and the remaining atomic constraints, is given as:

$$\begin{aligned} \{\, \{\, &([\alpha \mapsto \mathtt{Fun1<}\beta_1,\ \gamma_1\mathtt{>}, \epsilon \mapsto \mathtt{Fun1<}\beta_2,\ \gamma_2\mathtt{>}],\\ &\{\, \beta_1 \lessdot_? \beta, \gamma \lessdot_? \gamma_1, \beta_2 \lessdot_? \beta_1, \gamma_1 \lessdot_? \gamma_2 \,\})\,\}\,\} \end{aligned}$$

As in the above examples we annotate f by the type Fun1<Sub, Sub>. This means we instantiate Sub for β_1 and γ_1, too. Then are allowed to be instantiated ? super Sub for β_2 and ? extends Super for γ_2, as Sub $\lessdot_?$? super Sub and Sub $\lessdot_?$? extends Super. This means the result constraints of the unification are fulfilled.
This type corresponds to the inferred type in the example of the strawman approach. This means the desired result can also be inferred by this algorithm.
But is allowed to be instantiated ? extends Super for β_2 and ? super Sub for γ_2, as in this case the result constraints of the unification are fulfilled, too. This result corresponds to the less reasonable typing in Fig. 6.
This example illustrates that the type inference for Java 8 infers the desired types, but it is also possible to infer less reasonable typings. This result is not surprising, as these typings are correct Java 8 typings and the type inference algorithm has been proven to be sound and complete.

5 DISCUSSION AND DECISION

We considered the realisation of lambda expressions in Java 8 (Section 2), in the strawman approach's language Java$_\lambda$ (Section 3.1), and in Scala (Section 3.2). In addition we considered our type inference algorithms for a core of Java$_\lambda$ (Section 4.1) and a core of Java 8 (Section 4.2). Implementing function types in Java 8 needs additional type-casts for implementing direct lambda expression applications and for declaring types of higher-order functions (Section 2.2). Furthermore, the implementation in Java 8 allows less

reasonable subtyping relations between function types (Section 2.3). The biggest benefit of implementation in Java 8 is the possibility of operating existing callback functions by lambda expressions (Section 2.1). The problems highlighted are solved in Java$_\lambda$ retaining the possibility to implement callback functions by lambda expressions. In contrast, Java$_\lambda$ contains nominal types and structural function types that add complexity to the type system. However, Scala solves the type system complexity by introducing declared-site variance function traits (interfaces) as function types such that no mixture of structural function types and nominal types are contained in Scala. On the other hand, Scala allows no implementation of callback functions by lambda expressions.
These arguments led us to the decision to add Scala declaration-site variance function types to Java-TX so as to retain the possibility of implementing callback functions by lambda expressions. Furthermore, we adapted the type inference algorithm, such that *global* type inference is added. In other words, Java-TX is the language Java$_\lambda$, where we replaced the structural function types with Scala function types and added a type inference algorithm.

In the subsequent sections we will describe the implementation in Java-TX. First, in Section 6, we will portray the integration of functional interfaces as target types of lambda expressions and the Scala function types. Then, in Section 7, we will describe the adaptation of the type inference algorithm. The main adaptation has to be untertaken in the type unification, a process which we will detail in Section 8.

6 INTEGRATION OF FUNCTIONAL INTERFACES AND SCALA-LIKE FUNCTION TYPES

We extended Java 8 by two sets of special functional interfaces

```
FunN*<-A1, ... -AN, +R>{
  R apply(A1 arg1, ... AN);
}

FunVoidN*<-A1, ... -AN>{
  R apply(A1 arg1, ... AN);
}
```

As in Scala the "+"– and "-"–symbols declare the variance of the type parameters. "+" represents covariance and "-" represents contravariance. For FunN* and FunVoidN* types respectively, no use-site variance, i.e. wildcards, are allowed.
The following two definitions adds the special functional interfaces to the Java specification. First, we defined the type of a lambda expression. Then we extended the definition of compatibility with a target type.
These definitions extend the specification of Java 8 for lambda expressions (§15.27.3 [6]).

Definition 6.1 (Type of a lambda expression). Let a lambda expression with N formal parameters be given.

- If the lambda expression's parameters are explicitly typed, let its formal parameter types be given as $\theta_1, \ldots, \theta_N$.
- If the lambda parameters are assumed to have the types $\theta_1, \ldots, \theta_N$, then:
  - If the lambda body is either a statement expression (§14.8 [6]) or a void-compatible block, then the *type of a lambda expression* is defined as FunVoidN*<$\theta_1, \ldots, \theta_N$>.
  - If either the lambda body is an expression and has the type θ_0, or the lambda body is a value-compatible block, and each result expression (§15.27.2 [6]) has the type θ_0, then *the type of the lambda expression* is defined as FunN*<$\theta_1, \ldots, \theta_N$,$\theta_0$>.

This definition solves the problems described in Section 2.2 and 2.3. For direct lambda expression applications type-casts are no longer necessary. The example from Section 2.2 can be written without type-casts:

```
(x1->x2->. . .->xN->h(x1,. . . ,XN))
 .apply(a1).apply(a2). . . . .apply(aN)
```

The problem presented in Section 2.3 has also been solved. In Fig. 9 f typed by Fun2*<Number, Number,Integer> is assigned to

```
class MathStruc<A> {

 A model;

 Fun1*<Fun2*<A,A,A>,
       Fun1*<MathStruc<A>,MathStruc<A>>>
 innerOp = (o) -> (ms) ->
  new MathStruc<>(o.apply(this.model,ms.model));

 Fun2*<Number,Number,Integer> f = ...

 MathStruc(A m) {
     model =m;
 }

 void m() {
  MathStruc<Integer> msi = new MathStruc<>(1);
  MathStruc<Integer> res =
    msi.innerOp.apply(f)
                .apply(new MathStruc<>(3));
    }
}
```

Figure 9: MathStruc with Scala-like function types

innerOp's parameter o which is typed by Fun2*<Integer, Integer, Integer> . The type of f is a subtype of the type of o.
Introducing function types allows us to define the compatibility of any expression typed by a Fun$N$*–type with a target type. This is an extension compared to Def. 2.4, where only lambda expression can be compatible with target types.

Definition 6.2 (Expressions compatible with a target type).
An expression, which has the type FunVoidN*<$\theta_1, \ldots, \theta_N$> and FunN*<$\theta_1, \ldots, \theta_N$,$\theta_0$> respectively, is *compatible* in an assignment, invocation, or casting context with a *target type* T if T is a functional interface type and the expression is congruent with the function type of the ground target type derived from T, which means

- The function type has no type parameters.

- The number of parameter types of the function type is N.
- The parameter types of the function type are $\theta_1, \ldots, \theta_N$.
- If the result of the function type is void, then the type of lambda expression is FunVoidN*<$\theta_1, \ldots, \theta_N$>.
- If the result of the function type is a (non-void) type θ_0, then the type of lambda expression is FunN*<$\theta_1, \ldots, \theta_N$,$\theta_0$>.

This definition has an interesting consequence, as in the next example serves to illustrate.

Example 6.3. We will now consider again the JavaFX example from the end of Section 2.1. An additional local variable helloworld is assigned a lambda expression, which implements the callback function.

```
Fun1*<ActionEvent, String> helloworld
   = event -> System.out.println("Hello␣World!");
Button btn = new Button();
btn.setText("Say␣'Hello␣World'");
btn.setOnAction(helloworld);
```

The method setOnAction is called with argument helloworld and no longer with the lambda expression as its argument. As the type of helloworld is Fun1*<ActionEvent,String> , helloworld is compatible with the target type EventHandler<ActionEvent>. In Java 8 the local variable helloworld has to be typed by EventHandler<ActionEvent>.

7 ADAPTION OF THE TYPE INFERENCE ALGORITHM

Fig. 10 presents the main function **TI** of the type inference algorithm. As in [16] we have left methods out of consideration, as the emphasis in this paper is on lambda expressions. For a consideration of methods please refer to [19].
In **TI** first the function **TYPE** determines a set of constraints (subtyping-inequations). Furthermore, types (at most fresh type variables) are inserted into the abstract syntax tree. In the function **SOLVE** these constraints are solved.

```
$\textbf{TI}$: (TypeAssumptions, Class) $\to$ { (Constraints, TClass) }
$\textbf{TI}(\,Ass, \text{Class}(\,\tau, \text{extends}(\,\tau'\,), fdecls\,)\,) =$
  let $\underline{(\text{Class}(\,\tau, \text{extends}(\,\tau'\,), fdecls_t\,), ConS)} =$
        $\textbf{TYPE}(\,Ass, \text{Class}(\,\tau, \text{extends}(\,\tau'\,), fdecls\,)\,)$
      $\underline{\{\,(cs_1, \sigma_1), \ldots, (cs_n, \sigma_n)\,\}} = \textbf{SOLVE}(\,ConS\,)$
  in $\{\,(cs_i, \sigma_i(\,\text{Class}(\,\tau, \text{extends}(\,\tau'\,), fdecls_t\,)\,))\mid 1 \leqslant i \leqslant n\,\}$
```

Figure 10: Type inference algorithm's main function

```
$\textbf{TYPE}$: (TypeAssumptions, Class)
                                   $\to$ (TClass, ConstraintsSet)
$\textbf{TYPE}(\,Ass, \text{Class}(\,\tau, \text{extends}(\,\tau'\,), fdecls\,)\,) = \textbf{let}$
  $\underline{fdecls} = [\text{Field}(\,f_1, lexpr_1\,), \ldots, \text{Field}(\,f_n, lexpr_n\,)]$[5]
  $\underline{ftypeass} = \{\,\texttt{this}.f_i : a_i \mid a_i\ \textit{fresh type variables}\,\}$
     $\cup \{\,\texttt{this} : \tau, \texttt{super} : \tau'\,\} \cup \{\,\textit{visible types fields of}\ \tau'\,\}$
  $\underline{AssAll} = Ass \cup ftypeass$
  Forall $1 \leqslant i \leqslant n : \underline{(lexpi_{i_t} : rtyF_i, ConSF_i)} =$
     $\textbf{TYPEExpr}(\,AssAll, lexpr_i\,)$
  $\underline{fdecls_t} = [\text{Field}(\,a_1, f_1, lexpr_{1_t} : rtyF_1\,)$
              $, \ldots,$
              $\text{Field}(\,a_n, f_n, lexpr_{n_t} : rtyF_n\,)]$
 in
   $(\text{Class}(\,\tau, \text{extends}(\,\tau'\,), fdecls_t\,),$
   $(\bigcup_i ConSF_i \cup \{\,(rtyF_i \lessdot a_i) \mid 1 \leqslant i \leqslant n\,\}))$
```

Figure 11: The function TYPE

```
$\textbf{TYPEExpr}(\,Ass, \text{Lambda}(\,(x_1, \ldots, x_N), expr|stmt\,)\,) =$
  let $\underline{AssA} = \{\,x_i : a_i \mid a_i\ \textit{fresh type variables}\,\}$
    $\underline{(expr_t : rty, ConS)} = \textbf{TYPEExpr}(\,Ass \cup AssA, expr\,)$
   | $\underline{(stmt_t : rty, ConS)} = \textbf{TYPEStmt}(\,Ass \cup AssA, stmt\,)$
  in if $(rty == \texttt{void})$ then
    $(\text{Lambda}(\,(x_1 : a_1, \ldots, x_N : a_N), stmt_t : rty\,)$
     $: \text{FunVoidN*<}a_1, \ldots, a_N\text{>}, ConS)$
   else
    $(\text{Lambda}(\,(x_1 : a_1, \ldots, x_N : a_N), expr_t : rty | stmt_t : rty\,)$
     $: \text{FunN*<}a_1, \ldots, a_N, a\text{>}, ConS \cup \{\,rty \lessdot a\,\}),$
     where $a$ is a fresh type variable
```

Figure 12: TYPEExpr for lambda expressions

The function **TYPE** (Fig. 11) calls for each expression, which is assigned to a field, the function **TYPEExpr**. For each sub-expression of the respective expressions, **TYPEExpr** builds the contraints and inserts its types into the abstract syntax tree. For each form of possible expression, a function declaration of **TYPEExpr** is given, each of which are selected during runtime by pattern matching. These functions are unchanged in comparison to [14] except the declaration for the lambda expression. The new declaration is given in Fig. 12. The FunN functional interfaces are replaced by new FunN* and FunVoidN* interfaces.
We consider now two examples to illustrates the **TYPE** algorithm. First we present the MathStruc example. This example shows the the great benefit in code readability if there is type inference.

Example 7.1. Let the program be given as follows.

```
class MathStruc<A> {

  A model;
  innerOp = o -> ms ->
    new MathStruc<A>(o.apply(this.model,
                             ms.model));
  MathStruc(A m) { model=m; }
}
```

If we compare this declaration of innerOp especially with the declaration in Fig. 3 we recognize that type inference increases readability of programs enormously. The **TYPE** function inserts the following types[6]

```
innerOp:$\alpha$ =
  (o:$\beta$ -> (ms:$\gamma$ ->
    (new MathStruc<A>(o.apply(this.model:A,
                              ms.model:$\delta$):$\epsilon$
```

[5]Without loss of generality we assume that all fields are declared typeless and that all fields are initialized by expressions.

[6]We leave out some obvious type annotations.

```
    )):MathStruc<A>):Fun1*<γ,η>
  ):Fun1*<β,ζ>;
```

and determines the following set of constraints:

$$\{\, \texttt{Fun1*<}\beta,\zeta\texttt{>} \lessdot \alpha,\ \texttt{Fun1*<}\gamma,\eta\texttt{>} \lessdot \zeta,\ \texttt{MathStruc<A>} \lessdot \eta,\ \gamma \lessdot \texttt{MathStruc<}\delta\texttt{>},\ \epsilon \lessdot \texttt{A},\ \beta \doteq \texttt{Fun2*<}\iota,\kappa,\epsilon\texttt{>},\ \texttt{A} \lessdot \iota,\ \delta \lessdot \kappa \,\}.$$

The second example shows *global* type inference.

Example 7.2. Given the faculty function that we considered in Section 3.2 for Scala,

```
class Faculty {
  static fact[7];

  m () {
   fact = x -> if (x <= 1) return 1
               else return x * fact.apply(x-1);}
}
```

The **TYPE** algorithm inserted the following types[8]

```
class Faculty {
  static fact:α;

  β m () {
    fact:α =
      (x:γ ->
        (if (x:γ <= 1:Integer):Boolean
           (return 1):Integer
         else
           (return (x:γ *
                   (fact:α.apply(x-1)):δ):Integer
           ):Integer
        ):Integer
      ):Fun1*<γ,ε>;
  }
}
```

and determines the following set of constraints:

$$\{\, \texttt{Fun1*<}\gamma,\epsilon\texttt{>} \lessdot \alpha,\ \texttt{Integer} \lessdot \epsilon,\ \gamma \lessdot \texttt{Integer},\ \delta \lessdot \texttt{Integer},\ \alpha \doteq \texttt{Fun1*<}\phi,\delta\texttt{>}, \texttt{Integer} \lessdot \phi \,\}.$$

At the end of the next section we will present the application of the function **SOLVE** for both the above examples.
The function **SOLVE** is presented in Fig. 13. In **SOLVE** the type unification **TUnify** is called. The result of **TUnify** is a set of constraint pairs. In **SOLVE** the result pairs are filtered. Either pairs, which are in solved form (all pairs consist of $v \doteq \theta$ where v is type variable) or else additional pairs $v\,R\,v'$, where v' is also a type variable, are correct. For all other results the algorithm fails.
In comparison to [13] the type unification is extended. This extension will be considered in the next section.

8 ADAPTION OF THE UNIFICATION

The type unification problem for Java 8 types is given as: For a set of type term pairs

$$\{\, \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \,\}$$

[7]As Java 8 self-referencing in an initializer is disabled and a local variable which is assigned a lambda expression must be effectively final, `fact` is declared as a field.
[8]We leave out again some obvious type annotations.

```
SOLVE: $\texttt{ConstraintsSet} \longrightarrow \{\,\texttt{ConstraintsSet} \times Subst\,\}$
SOLVE($ConsS$) = let $\underline{subs}$ = TUnify($ConsS$) in
  if (there are $c \in subs$ in solved form) then
    $\{\, (\emptyset, \{\, v \mapsto \theta \,\}) \mid (v \doteq \theta) \in c,\ c \in subs$ is in solved form $\}$
  if (there are $c \in subs$, which has the form
    $\{\, v\,R\,v' \mid v, v'$ are type vars $\} \cup$
    $\{\, v \doteq \theta \mid v$ is a type var $\})$
  then
    $\{\, (\{\, v\,R\,v' \}, \{\, v \mapsto \theta \,\}) \mid (v\,R\,v') \in c, (v \doteq \theta) \in c,$
                    $c \in subs$ has the given form $\}$
  else fail
```

Figure 13: The function SOLVE

a substitution σ, called unifier, is necessary, such that

$$\sigma(\,\theta_i\,) \leq^* \sigma(\,\theta'_i\,),\ \ \forall 1 \leqslant i \leqslant n.$$

In [13] we proved that for Java types with wildcards but without multiple inheritance the unification problem is indeed not unitary but finitary. This means there is a finite number of most general unifiers.
In this paper we extend the type unification by the new function types `FunN*` and `FunVoidN*`. The rules for `FunN*` are presented in Fig. 14. The corresponding `FunVoidN*` rules are given analogously, where the respective result types θ_0, θ'_o, b_0, and b'_0 are left out.

Theorem 8.1. *The type unification algorithm from [13], extended by the unification rules for the* `FunN*` *and the* `FunVoidN*` *function types is sound and complete.*

Proof. As we have proved the soundness and completeness of the type unification algorithm in [13], for all new rules we now have to show the property: If and only if a substitution is a most general unifier before applying a type unification rule, is it also a most general unifier after applying the type unification rule.

redFunN rule.* From the subtyping definition of the `FunN*` function types follows, for a unifier σ

$$\sigma(\,\texttt{FunN*<}\theta'_1, \ldots, \theta'_n,\theta_o\texttt{>}\,) \leq^* \sigma(\,\texttt{FunN*<}\theta_1, \ldots, \theta_N,\theta'_0\texttt{>}\,)$$

is valid if and only if $\sigma(\,\theta_o\,) \leq^* \sigma(\,\theta'_0\,)$ is valid and for $1 \leqslant i \leqslant N$: $\sigma(\,\theta_i\,) \leq^* \sigma(\,\theta'_i\,)$ is valid. This means if and only if σ is a most general unifier before applying the **redFunN*** rule it is also a most general unifier after applying the rule.

grFunN rule.*

$$\sigma(\,\texttt{FunN*<}\theta'_1, \ldots, \theta'_N,\theta_o\texttt{>}\,) \leq^* \sigma(\,a\,)$$

is valid if and only if

$$\sigma \in \{\, a \mapsto \tau \mid \texttt{FunN*<}\theta'_1, \ldots, \theta'_N,\theta_o\texttt{>} \leq^* \tau \,\}.$$

As $\texttt{FunN*<}\theta'_1, \ldots, \theta'_N,\theta_o\texttt{>} \leq^* \tau$ is valid if and only if $\tau = \texttt{FunN*<}\theta_1, \ldots, \theta_N,\theta'_o\texttt{>}$ with $\theta_0 \leq^* \theta'_0$ and for $1 \leqslant i \leqslant N$: $\theta_i \leq^* \theta'_i$, this is equivalent to

$$\sigma \in \{\, a \mapsto \texttt{FunN*<}b_1, \ldots, b_N,b'_o\texttt{>},\ b'_o \mapsto \theta'_0, b_1 \mapsto \theta_1, \ldots, b_N \mapsto \theta_N \mid \theta_0 \leq^* \theta'_0, \theta_i \leq^* \theta'_i,\ \text{for } 1 \leqslant i \leqslant N. \,\}$$

$$(\text{redFunN*})\ \frac{Eq \cup \{\ \texttt{FunN*<}\theta'_1,\ldots,\theta'_N\texttt{,}\theta_o\texttt{>}\ \lessdot\ \texttt{FunN*<}\theta_1,\ldots,\theta_N\texttt{,}\theta'_o\texttt{>}\ \}}{Eq \cup \{\ \theta_0 \lessdot \theta'_0, \theta_1 \lessdot \theta'_1, \ldots, \theta_N \lessdot \theta'_N\ \}}$$

$$(\text{grFunN*})\ \frac{Eq \cup \{\ \texttt{FunN*<}\theta'_1,\ldots,\theta'_N\texttt{,}\theta_o\texttt{>}\ \lessdot\ a\ \}}{Eq \cup \{\ a \doteq \texttt{FunN*<}b_1,\ldots,b_N\texttt{,}b'_o\texttt{>}, \theta_0 \lessdot b'_0, b_i \lessdot \theta'_i,\ \text{for}\ 1 \leqslant i \leqslant N\ \}}$$
where $b'_0, b_1, \ldots, b_N$ are fresh type variables

$$(\text{smFunN*})\ \frac{Eq \cup \{\ a\ \lessdot\ \texttt{FunN*<}\theta_1,\ldots,\theta_N\texttt{,}\theta'_o\texttt{>}\ \}}{Eq \cup \{\ a \doteq \texttt{FunN*<}b'_1,\ldots,b'_N\texttt{,}b_o\texttt{>}, b_0 \lessdot \theta'_0, \theta_i \lessdot b'_i,\ \text{for}\ 1 \leqslant i \leqslant N\ \}}$$
where $b_0, b'_1, \ldots, b'_N$ are fresh type variables

Figure 14: Java type unification rules for the FunN* function types

And this is equivalent to: For all σ

$$\sigma(\,a\,) = \sigma(\,\texttt{FunN*<}b_1,\ldots,b_N\texttt{,}b'_o\texttt{>}\,), \sigma(\,\theta_0\,) \leq^* \sigma(\,b'_0\,),\ \text{and}$$
$$\sigma(\,b_i\,) \leq^* \sigma(\,\theta'_i\,)\ \text{for}\ 1 \leqslant i \leqslant N$$

is valid. This means if and only if σ is a most general unifier before applying the **grFunN*** rule it is also a most general unifier after applying the rule.

smFunN rule.* For the **smFunN*** rule the condition follows analogous to the **grFunN*** rule. □

We would like to end this section with three further examples. First of all, we will present the solution for the example from Section 4.2. Then we will present the applications of **SOLVE** for Examples 7.1 and 7.2.

Example 8.2. For the example from Section 4.2 the result of **TYPE** is the program with inserted types

```
void m() {
  f:α = ((x:β) -> x:γ):Fun1*<β, γ>;
  var g:ϵ;
  g:ϵ = f:α;
}
```

and the set of type constraints:

$$\{\ \texttt{Fun1<}\beta,\ \gamma\texttt{>} \lessdot \alpha, \alpha \lessdot \epsilon\ \}.$$

Applying the type unification rule **grFunN*** twice, we get

$$\{\ \alpha \doteq \texttt{Fun1*<}\beta_1,\ \gamma_1\texttt{>}, \epsilon \doteq \texttt{Fun1*<}\beta_2,\ \gamma_2\texttt{>}$$
$$\beta_1 \lessdot \beta, \gamma \lessdot \gamma_1, \beta_2 \lessdot \beta_1, \gamma_1 \lessdot \gamma_2\ \}$$

From this follows as result of **SOLVE**:

$$\{\ \{\ ([\alpha \mapsto \texttt{Fun1*<}\beta_1,\ \gamma_1\texttt{>}, \epsilon \mapsto \texttt{Fun1*<}\beta_2,\ \gamma_2\texttt{>}],$$
$$\{\ \beta_1 \lessdot \beta, \gamma \lessdot \gamma_1, \beta_2 \lessdot \beta_1, \gamma_1 \lessdot \gamma_2\ \})\ \}\ \}$$

Once again we annotate f by the type Fun1<Sub, Sub>. This means we instantiate Sub for β_1 and γ_1, too. Now, we are allowed to instantiate Sub for β_2 and Sub as well as Super for γ_2, such that the result constraints of the unification are fulfilled. This means that for g only the expected types Fun1*<Sub,Sub> and Fun1*<Sub,Super> are allowed.

Example 8.3. In the following we complete Example 7.1 by applying **SOLVE**. At first, the type unification rule **grFunN*** is applied twice:

$$\begin{aligned}\{\ &\alpha \doteq \texttt{Fun1*<}\beta_1\texttt{,}\zeta_1\texttt{>},\ \beta_1 \lessdot \beta,\ \zeta \lessdot \zeta_1\\ &\zeta \doteq \texttt{Fun1*<}\gamma_1\texttt{,}\eta_1\texttt{>},\ \gamma_1 \lessdot \gamma,\ \eta \lessdot \eta_1,\\ &\texttt{MathStruc<A>} \lessdot \eta, \gamma \lessdot \texttt{MathStruc<}\delta\texttt{>},\\ &\epsilon \lessdot \texttt{A},\ \beta \doteq \texttt{Fun2*<}\iota,\kappa\texttt{,}\epsilon\texttt{>},\ \texttt{A} \lessdot \iota,\ \delta \lessdot \kappa\ \}.\end{aligned}$$

In the following we will condense some steps of substitutions and sub- and supertype constructions. In these steps the set of inequations are multiplied. Then further on, we will consider just one representative of this set, where no wildcards are inferred.

$$\begin{aligned}\{\ &\alpha \doteq \texttt{Fun1*<Fun2*<A,}\ \delta\texttt{,A>,}\zeta_1\texttt{>},\\ &\texttt{Fun1*<MathStruc<}\delta\texttt{>,MathStruc<A>>} \lessdot \zeta_1\\ &\zeta \doteq \texttt{Fun1*<MathStruc<}\delta\texttt{>,MathStruc<A>>},\\ &\eta \doteq \texttt{MathStruc<A>},\ \gamma \doteq \texttt{MathStruc<}\delta\texttt{>},\\ &\epsilon \doteq \texttt{A},\ \beta \doteq \texttt{Fun2*<A,}\ \delta\texttt{,A>}\ \}^9.\end{aligned}$$

Finally, the inequation Fun1*<MathStruc<δ>,MathStruc<A>> $\lessdot\ \zeta_1$ is solved by the **grFunN*** rule. With some sub- and supertype constructions and some substitutions we get the result of **SOLVE**:

$$\begin{aligned}(\emptyset, [&\alpha \mapsto \texttt{Fun1*<Fun2*<A,}\ \delta\texttt{,A>,}\\ &\qquad\texttt{Fun1*<MathStruc<}\delta\texttt{>,MathStruc<A>>>},\\ &\zeta \mapsto \texttt{Fun1*<MathStruc<}\delta\texttt{>,MathStruc<A>>},\\ &\eta \mapsto \texttt{MathStruc<A>},\ \gamma \mapsto \texttt{MathStruc<}\delta\texttt{>},\\ &\epsilon \mapsto \texttt{A},\ \beta \mapsto \texttt{Fun2*<A,}\ \delta\texttt{,A>}])^9.\end{aligned}$$

We infer a free type variable δ that becomes another type parameter of the class MathStruc. In Fig. 15 we present the program with the inserted types.
It is possible to avoid the free type variables δ. For this ms to have been explicitly typed:

```
innerOp = o -> (MathStruc<A> ms) -> ...
```

Finally, we will present the result of the faculty example.

Example 8.4. Now let us turn to completing Example 7.2. With the substitution of α in Fun1*<γ,ϵ> $\lessdot\ \alpha$. by $\alpha \doteq$ Fun1*<ϕ,δ> we get Fun1*<γ,ϵ> $\lessdot$ Fun1*<ϕ,δ>. with the **redFunN*** rule, we get $\phi \lessdot \gamma$ and $\epsilon \lessdot \delta$. With some sub- and supertype constructions and some substitutions we get

$$\begin{aligned}\{\ (\emptyset, [&\gamma \mapsto \texttt{Integer},\ \phi \mapsto \texttt{Integer},\ \epsilon \mapsto \texttt{Integer},\qquad .\\ &\delta \mapsto \texttt{Integer},\ \alpha \mapsto \texttt{Fun1*<Integer,Integer>},\ ])\ \}\end{aligned}$$

[9]We have omitted all substituted and all type variables that are never again required.

```
class MathStruc<A,Delta> {

 A model;

 Fun1*<Fun2*<A,Delta,A>,
       Fun1*<MathStruc<Delta>,MathStruc<A>>
 innerOp = (Fun2*<A,Delta,A> o ->
   (MathStruc<A, Delta> ms) ->
     new MathStruc<A,Delta>(o.apply(this.model,
                                      ms.model));
}
```

Figure 15: Inserted inferred types

9 CONCLUSION AND FUTURE WORK

In this study we considered the introduction of Scala-like function types into Java-TX. The main additional feature of Java-TX in comparison to Java 8 is *global* type inference. Until now Java-TX had a type inference algorithm, which infers all correct typings for a typeless Java 8 program. We showed that any indeed correct typings are less reasonable.

In order to solve this problem we considered the approach of functional interfaces as target types of lambda expression, given in Java 8, in detail. We showed its main benefit, i.e. the possibility of implementing callback functions in many existing libraries. But we also demonstrated the disadvantages of this approach, which are the unbeautiful realisation of direct lambda expression applications and subtypes of function types. Furthermore, we considered two other approaches, the strawman approach and the approach of function types in Scala. And, finally we discussed the different approaches and showed that it is indeed a good idea to retain the possibility of implementing callback functions by lambda expressions and add Scala-like function types.

To this purpose we have redefined the language specification through the introduction of real function types for lambda expressions. In addition, we extended the definition of compatibility with target types.

As a final last step, we adapted our type inference algorithm. The main adaptation had to be undertaken in the type unification, where three new rules are introduced in order to handle the added function types.

If we compare our approach of function types in Java with the arguments of [4] against real function types in Java 8 (cp. Section 2), we will see that three of the four given arguments, are in fact no longer given. More specifically, there is no mixture of structural and nominal types, as the function types are special functional interfaces. As we allow lambda expressions to implement callback functions futher on, no divergence in library styles arises, i.e. some libraries would continue to use callback interfaces, while others would use structural function types. Moreover, the syntax is not unwieldy as it had been in the strawman approach.

Only the runtime representation of function types is limited by type erasure so that it would not be possible to overload methods `m(T->U)` and `m(X->Y)`.

In future work we will circumvent this problem. In the programming language PIZZA we saw two different translation approaches, a homogenous and a heterogenous [8]. In Java only the homogenous translation has been introduced so as to erase the type parameters. We will renew the heterogenous approach in order to avoid type erasures so that overloading in the above style will become possible. Furthermore, we will optimize the function **SOLVE**, especially the type unification. At the present time there are somes cases being computed where it is unnecessary.

REFERENCES

[1] Hendrik P. Barendregt. 1984. *The lambda calculus : Its syntax and semantics.* North-Holland, Amsterdam.

[2] C# 2012. C# Language Specification Version 5.0. (2012). https://download.microsoft.com/download/0/B/D/0BDA894F-2CCD-4C2C-B5A7-4EB1171962E5/CSharp%20Language%20Specification.docx

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

[4] Brian Goetz. 2013. State of the Lambda. (September 2013). http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html

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

[6] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. *The Java® Language Specification* (Java SE 8 ed.).

[7] Lambda. 2010. Project Lambda: Java Language Specification draft. (2010). http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt Version 0.1.5.

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

[9] Martin Odersky and Philip Wadler. 1997. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97).* ACM, New York, NY, USA, 146–159. https://doi.org/10.1145/263699.263715

[10] Martin Odersky, Matthias Zenger, and Christoph Zenger. 2001. Colored local type inference. *Proc. 28th ACM Symposium on Principles of Programming Languages* 36, 3 (2001), 41–53.

[11] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

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

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

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

[15] Martin Plümicke. 2012. Functional implementation of well-typings in Java$_\lambda$. In *Draft proceedings of the 24th symposium on implementation and application of functional languages (IFL 2012)*, Ralf Hinze (Ed.). Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, 403–415.

[16] Martin Plümicke. 2015. More Type Inference in Java 8. In *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers (Lecture Notes in Computer Science)*, Andrei Voronkov and Irina Virbitskaite (Eds.), Vol. 8974. Springer, 248–256.

[17] Scala 2017. The Scala Language Specification. (2017). http://www.scala-lang.org/files/archive/spec/2.12 Version 2.12.

[18] Andreas Stadelmeier. 2015. Java type inference as an Eclipse plugin. In *45. Jahrestagung der Gesellschaft für Informatik, Informatik 2015, Informatik, Energie und Umwelt, 28. September - 2. Oktober 2015 in Cottbus, Deutschland.* 1841–1852. http://subs.emis.de/LNI/Proceedings/Proceedings246/article18.html

[19] Andreas Stadelmeier and Martin Plümicke. 2015. Adding overloading to Java type inference. In *Tagungsband der Arbeitstagung Programmiersprachen (ATPS 2015)*, Vol. Vol-1337. CEUR Workshop Proceedings (CEUR-WS.org), 127–132.

[20] Bjarne Stroustrup. 2013. *The C++ Programming Language* (fourth ed.). Addison-Wesley.