Functional Interfaces vs. Function Types in Java with Lambdas[*]

– Extended Abstract –

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

1 Introduction

In several steps the Java type system is extended by features, which we know from functional programming languages. In Java 5.0 [GJSB05] generic types as well as a limited form of existential types (bounded wildcards) are introduced. In Java 8 the language will be expanded by *lambda expressions*, *functional interfaces as target types*, *method and constructor references* and *default methods*. In an earlier approach real function types were introduced as types of lambdas. The function types have been replaced by functional interfaces (interfaces with one method).

Brian Goetz [Goe13] gave some reasons for this decision, why no function types have been introduced: *the mix of structural and nominal types*, *the divergence of library styles – some libraries would continue to use callback interfaces, while others would use structural function types*, *the syntax could be unwieldy*, *there would not be a runtime representation for each distinct function type, caused by type erasure, e.g, it would not be possible (perhaps surprisingly) to overload methods* `m(T->U)` *and* `m(X->Y)`.

This decision allows, however, to use lambda expressions as an abbreviation for anonymous inner classes. In this paper we will show some disadvantages and propose an idea to allow both function types and functional interfaces.

2 Functional interfaces vs. function types

First we will present the idea of functional interfaces as target types of lambda expressions. A lambda expression in Java 8 has no explicit type. The type is inferred 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";
```

There is a canonical target type of a lambda expression `(T1 a1, ..., TN aN) -> lbody` `interface FunN<R,T1 , ..., TN> {R apply(T1 arg1 , ..., TN argN);}`

where the type variables `T1` , . . . , `TN` are instanced, respectively.

Using functional interfaces as function types there are two disadvantages wrt. subtyping and direct evaluation of lambda expressions:

Subtyping: While for real function types there is a subtyping relation

$$T'_1 \times \ldots \times T'_N \to T_0 \leq^* T_1 \times \ldots \times T_N \to T'_0,\ T_i \leq^* T'_i.$$

for functional interfaces this holds only by introducing wildcards

`FunN<`$T_0, T'_1, \ldots, T'_N$`>` $\leq^*$ `FunN<? extends` T'_0`, ? super` $T_1, \ldots,$ `? super` T_N`>`, $T_i \leq^* T'_i$.

Evaluation of lambda expressions: In the λ-calculus a lambda expression can be applied directly to its arguments like: $((x_1, , \ldots, x_N) \to h(x1, , \ldots, xN))(a_1 \ldots, a_N)$.

In Java 8 this is only possible by applying the method of the functional interface and introducing a type-cast

```
((FunN<T0,T1,...,TN>)(x1, ..., xN)->h(x1, ..., xN)).apply(a1...,aN);
```

For curried functions this becomes very unbeautiful:

```
((Fun1<Fun1<Fun1<...Fun1<T0,TN>,...>,T2>,T1>)
 ((x1) -> (x2) ->...-> (xN) -> h(x1,...,XN))).apply(a1).....apply(aN));
```

3 Solution

We propose to extend Java 8 by real function types, such that lambda expressions have explicite types. Additionally we maintain the target typing of lambda expressions, which means that a function type is then converted automatically to a functional interface. Then Java would have the properties: *lambda expressions as abbreviations for anonymous inner classes*, *subtyping as expected for function types* and *the possibility to apply a lambda expression to its arguments, directly*.

References

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

[Goe13] Brian Goetz. State of the Lambda, September 2013.