Introducing Scala-like functional interfaces into Java

Martin Plümicke

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

Abstract. The Java type system has been extended twice. In Java 5.0 generics and wildcards are introduced. In Java 8 functional interfaces (interfaces with one abstract method) are defined as target types of lambda expressions. We have given in the past type inference algorithms for Java 5.0 as well as Java 8.
In this paper we propose to extend the Java type system again. We introduce real function types as explicit types for lambda expressions. We do this in a similar way to the Scala approach but without erasing the target typing mechanism for lambda expressions from Java 8.

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 the problem of target typing in this paper. For this we extend the Java type system again. We call this extension Java-TX (Java Type Extended).
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 relevantly 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.

1 Real function types

In the past we considered two different approaches of lambda expression typing for our type inference algorithm. While in [Plü11] real function types are

considered, in [Plü15] the Java 8-like functional interface are used. Also the first advisements of introducing lambda expressions into Java considers both approaches. While [LLB] favored functional interfaces (they called them SAM-types), [BGGvdA] and [CS] preferred real function types as types of lambda expressions (here called closures). Also Sun/Oracle favored in their *Project Lambda* until version 0.1.5 of the Java Language Specification draft [Lam10] real function types. Now, in Java 8 functional interfaces are implemented. In Java-TX we merge these both approaches, as both have some advantages.
Let us consider an example to explain the differences: The interface `Operation` and the function `doOperation` are given by

```
interface Operation {
  public int op (int x, int y);
}

void doOperation(Operation o) { ... }
```

To call the method `doOperation` in Java until version 7 typically an anonymous inner class is created, e.g.:

```
doOperation(new Operation () {
   public int op (int x, int y) { return x + y; } });
```

Lambda expressions in Java 8 simplified the call by using

```
doOperation((int x, int y) -> x + y)
```

Lambda expressions like this have no explicit types in Java 8, they receive a target type from the context. In this case the lambda expression `(int x, int y) -> x + y` gets the target type `Operation`.
The great benefit of this feature is the simplification of implementing call-back interfaces without anonymous inner class constructions. As there are many call-back interfaces in the Java standard library this feature facilitate Java programming, relevantly.

In [Lam10] explicit function types had been introduced, which meant, that

```
#int(int, int)
```

could be used instead of the interface declaration `Operation`. If we compare the Java 8 declaration of `doOperation`

```
void doOperation(Operation o) { ... }
```

with a declaration with function types

```
void doOperation(#int(int, int) o) { ... }
```

we will recognize the differences. The benefit of the approach with function types is, that the type of a lambda expression can be given directly. No additional interface has to be declared. Furthermore the meaning of the type is more obvious.

In Java-TX we unite both approaches.

The rest of the section is structured as follows. First we introduce Java interfaces Fun$N$, that simulates function types in Java 8. We present the problems of this approach. Second, we introduce special interfaces Fun$N*$, that correspond to the Scala function types. Third, we define the Fun$N*$–types as explicit lambda expression's types. Fourth, we extend the Fun$N*$–types to methods. Fifth, we show on this base how both concepts functional interfaces as compatible target types and real function types as types of lambda expressions can be integrated. Finally, we extend field declarations, such that free type variable for the fields can be defined.

1.1 The 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.

The determined types are called *target types*. Not all determined target types are correct. The determined target type is correct if the lambda expression is compatible with it [Goe13].
A lambda expression is *compatible* with a target type T, if

- T is a functional interface type.
- The lambda expression has the same number of parameters as T's method, and those parameters' types are the same.
- Each expression returned by the lambda body is compatible with T's method's return type.
- Each exception thrown by the lambda body is allowed by T's method's throws clause[1].

The lambda expression `() -> "done";` is compatible to the target types `Callable<String>` and `PrivilegedAction<String>`.
As there are many callback interfaces in the existing Java libraries this approach is very convenient as it avoids writing uncomfortable anonymous inner classes.

[1] Exceptions are not considered in this paper.

In [Plü14a] 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);
}
```

This approach solves the problem, that for the programmers the real type is hidden in the functional interface, but it has also some disadvantages. It is very difficult to use subtypes of functional interfaces. Furthermore the direct evaluation of lambda expressions is very inconvenient.

Subtyping For function types normally holds

$$(T'_1, \ldots, T'_N) \to T_0 \ \leq^* \ (T_1, \ldots, T_N) \to T'_0, \quad \text{for } T_i \leq^* T'_i$$

but in Java 8

$$\texttt{Fun}N\texttt{<}T_0, T'_1, \ldots, T'_N\texttt{>} \ \not\leq^* \ \texttt{Fun}N\texttt{<}T'_0, T_1, \ldots, T_N\texttt{>}, \text{ for } T_i \leq^* T'_i,$$

as neither covariant nor contravariant type arguments are possible. Alternatively, wildcards can simulate these properties. This means wildcards must be used for the subtyping property of function types:

$\texttt{Fun}N\texttt{<}T_0, T'_1, \ldots, T'_N\texttt{>} \ \leq^* \texttt{Fun}N\texttt{<?}\ \texttt{extends}\ T'_0, \texttt{?}\ \texttt{super}\ T_1, \ldots, \texttt{?}\ \texttt{super}\ T_N\texttt{>}$,
for $T_i \leq^* T'_i$.

The following example shows the inconvenience of this property.

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 wildcard 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.

Direct application of lambda expressions An direct application of a lambda expression to its arguments like this:

```
((x1,...,xN) -> h(x1,...,xN)).apply(arg1,...,argN);
```

is wrong, as the lambda expression has no type.
If a type-cast is inserted the lambda expression gets its target type form the context:

```
((FunN<T0, T1, ..., TN> )
 ((x1,...,xN) -> h(x1,...,xN))).apply(arg1,...,argN);
```

Then the application of the lambda expression is correct.

Summary of the problems and its solutions

- The loss of function types can be rectified by introducing Fun$N$–Interfaces.
- The Fun$N$–Subtyping problem can be rectified by using wildcards.
- The impossibility of direct application of lambda expressions can be rectified by using type-casts.

All problems are solvable, but the solutions are not beautiful.

1.2 The special interface Fun$N*$

Therefore in Java-TX we introduce a set of 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>[2] {
    R apply(T1 arg1, ..., TN argN);
}
```

where Fun$N*$<T_0,T'_1,...,T'_N> $\leq^*$ Fun$N*$<T'_0,T_1,...,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);
```

[2] The arguments are covariant resp. contravariant, written as in Scala [Ode14]

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>` .

1.3 Fun$N*$ as explicit type of lambda expressions

In Java 8 lambda expressions have no explicit type. In Java-TX the lambda expressions are typed by the corresponding `Fun`$N*$–types. This language extension allows to use lambda expressions without context, which is very important for the direct evaluation of lambda expressions. This means that lambda expressions like

```
((x1,...,xN) -> h(x1,...,xN)).apply(arg1,...,argN);
```

without type-casts are possible.

1.4 Fun$N*$ as types of methods

Consequently we give the methods also Fun$N*$–types. This means with the class `CL`

```
class Cl {

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

}
```

the method reference `CL::meth` has the type `Fun`$N*$`<`$\mathtt{T_o}$`,`$\mathtt{T_1}$`,...,`$\mathtt{T_N}$`>`.
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.

1.5 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 2. 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 lambda expressions fit the respective target types.

1.6 Fields and lambda expressions

For field declarations we do a small adaption. We allow to declare free type variables.
In Java 8 it is possible to declare e.g.:

```
Fun1<Integer,Integer> id = (x) -> x;
```

But no free type variables can be given for fields. We allow this in Java-TX. This means declarations like

```
<T> Fun1*<T,T> id = (x) -> x;
```

are possible.

2 Function types and type-erasures

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)`.

– Brian Goetz [Goe13] –

This citation from Brian Goetz is one of the most important arguments against introducing real function type into Java 8. The following example shows the problem. There are overloaded methods, where the arguments are different function types with the same number of arguments.

```
void apply(Fun1*<Integer, Integer> f) { ...}

void apply(Fun1*<Boolean, Boolean> f) { ...}
```

In Java 8 this is prohibited. The reason for this is that the parameters are erased during compilation. This means that the overloading is not resolvable during runtime, as both argument types are equal.
As this is for the developer certainly inapprehensible, we addressed this problem, such that generics can be introduced into byte-code [Plü14b].

2.1 Class loader

In this approach many class files are generated and all these classes have to be loaded into the JVM. This could lead to performance problems. There are some ideas to solve this problem. In [UTO13] types, where the type parameters are instantiated are defined as subtypes of the type without instantiated parameters. The methods are implemented as redirection to the original method, which is inherited. This means that the class files of instantiated type parameters are reduced to a minimum of code size. That would reduce the loading time.
In [ORW00] a heterogenous compilation is given, that preserves the type parameters, too. They describe a changed class loader, that needs only loading a part of classes, where the type parameters are instantiated. This class loader could make the class loading with a good performance.

3 Conclusion and outlook

3.1 Conclusion

We have presented an extension of the Java type system. We proposed to introduce real function types. We gave an approach similar to the approach in Scala. We showed how both concepts can be integrated. The concept of using functional interfaces as target types for lambda expressions, as well as our concept of real function types. So the advantages of both concepts can be used.
Type-erasure during compilation can render these extensions of the type system ineffectice. Therefore we gave an approach to introduce real generics in bytecode.
Oracle decided consciously against real function types in Java 8. Brian Goetz, the language architect, gave in [Goe13] four arguments. In the following we will show that our approach respects these arguments.

It would add complexity to the type system and further mix structural and nominal types (Java is almost entirely nominally typed).

Our approach introduces the `Fun`N`*`–types, that are also nominal types, such that structural and nominal types are not mixed.

It would lead to a divergence of library styles – some libraries would continue to use callback interfaces, while others would use structural function types.

As we integrate both concepts no divergence will arise. Even if new callback interfaces will use `Fun`N`*`–types implementations of both interfaces can be done by lambda expressions.

The syntax could be unwieldy, especially when checked exceptions were included.

As we have avoided to introduce a new function type syntax as in early versions of Java 8 (e.g. `#int(int)`), no unwieldy syntax is arised.

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)`*.*

This problem is solved by the introduction of real generics in byte-code. With this extension overloaded methods `m(T->U)` and `m(X->Y)` become possible.

3.2 Outlook

We have done prototypical implementations for both type system extensions. In the future we will integrate these extensions to our Java-TX system.
Another problem, that has to be solved is the class loading problem in the JVM. In original Java for each Java class one class file is generated. In contrast in the approach of this paper for each used parameter instance an own class files is generated. It could be possible that many class files are generated and loaded during runtime. As the method's byte-code instructions do not differ in all these class files, there is a potential to optimize the approach. In Section 2.1 we announced two possible approaches to solve these problems. This is to be done in the future.

References

[BGGvdA] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Closures for the java programming language (aka BGGA).

[CS] Stephen Colebourne and Stefan Schulz. First-class methods: Java-style closures (aka FCM).

[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™ Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.

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

[Lam10] Lambda. Project lambda: Java language specification draft, 2010. Version 0.1.5.

[LLB] Bob Lee, Doug Lea, and Josh Bloch. Concise instance creation expressions: Closures without complexity (aka CICE).

[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.

[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.

[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.

[Plü14a] 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.

[Plü14b] Martin Plümicke. Generic instances in java byte code. In *Programmiersprachen und Rechenkonzepte: 31. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, Bad Honnef, 28. – 30. April 2014*, Technische Berichte der TU Wien, 2014. (to appear).

[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.

[UTO13] Vlad Ureche, Cristian Talau, and Martin Odersky. Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. In *OOPSLA*, pages 73–92, 2013.