Introducing Scala-like functional interfaces into Java

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

6. Mai 2015

Overview

Lambda–Expressions in Java 8

```
(x) -> x;
```

Lambda–Expressions in Java 8

```
(x) -> x;
```

has no explicite type.

Lambda–expressions get functional interfaces as compatible target types from the environment.

Functional Interface

Interfaces with a single abstract method.

E.g.

```
interface Comparator<T>  { int compare(T x, T y); }
interface FileFilter     { boolean accept(File x); }
interface DirectoryStream.Filter<T> { boolean accept(T x); }
interface Runnable       { void run(); }
interface ActionListener { void actionPerformed(...); }
interface Callable<T>    { T call(); }
```

Functional interfaces as compatible target types of lambda expressions

A lambda expression is *compatible* with a 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

Target type (Example)

```
Identity id_fun = (x) -> x;

interface Identity {
    int id(int x) ;
}
```

Application:

```
...
System.out.println(id_fun.id(5));
...
```

Problems:

- The type's structure of id_fun is not visible.
- Method's (id's) type is not principal.
- The name of the method id is arbitrary.

Simulating function types

Canonical representation

Lemma: There is an equivalence class of compatible target types for a lambda expression. For the equivalence class of the compatible target types of a lambda expression, there is a canonical representation

$$\texttt{Fun}N\texttt{<}R, T1, \ldots, TN\texttt{>}$$

with

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

if the type of the single method of a compatible target type is

$$(T1, \ldots, TN) \rightarrow R$$

Subtyping

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

Subtyping

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

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

Subtyping

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

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

Example:

For `Integer` $\leq^*$ `Number` $\leq^*$ `Object` holds:

$$\texttt{Number} \to \texttt{Number} \;\leq^*\; \texttt{Integer} \to \texttt{Object}$$

Subtyping

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

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

Example:
For `Integer` $\leq^*$ `Number` $\leq^*$ `Object` holds:

$$\texttt{Number} \to \texttt{Number} \quad \leq^* \quad \texttt{Integer} \to \texttt{Object}$$

but

```
Fun1<Number,Number> f_NumNum = ...
Fun1<Object,Integer> f_IntObj = f_NumNum
```

is **wrong!**, as

$$\texttt{Fun1<Number,Number>} \quad \not\leq^* \quad \texttt{Fun1<Object,Integer>}$$

Subtyping with wildcards

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

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

Subtyping with wildcards

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

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

but

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

Subtyping with wildcards (Example):

Example:

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

Subtyping with wildcards (Example):

Example:

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

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

Subtyping with wildcards (Example):

Example:

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

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

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

Subtyping with wildcards (Example):

Example:

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

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

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

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

Syntax of function types with wildcards

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

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

Syntax of function types with wildcards

```
//A -> (B -> (((A, B) -> C) -> C)))
Fun1<? extends Fun1<? extends Fun1<? extends C,
                                   ? super Fun2<? extends C,
                                                ? super A,
                                                ? super B>>,
                         ? super B>,
      ? super A>
g = x -> y -> f -> f.apply(x,y);
```

Syntax of function types with wildcards

```
//A -> (B -> (((A, B) -> C) -> C)))
Fun1<? extends Fun1<? extends Fun1<? extends C,
                                   ? super Fun2<? extends C,
                                                ? super A,
                                                ? super B>>,
                         ? super B>,
      ? super A>
g = x -> y -> f -> f.apply(x,y);
```

Ugly syntax!

Direct application of lambda expressions

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

wrong!, as the lambda expression has no type

Direct application of lambda expressions

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

wrong!, as the lambda expression has no type

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

ok!, as there is cast–expression

Direct application of lambda expressions

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

wrong!, as the lambda expression has no type

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

ok!, as there is cast–expression

Currying: $f : T_1 \to T_2 \to \ldots \to T_N \to T_0$

```
((Fun1<Fun1<Fun1<...Fun1<T0,TN>,...>,T2>,T1> )
 (x1) -> (x2) ->...-> (xN) -> h(x1,...,XN))
 .apply(a1).apply(a2).....apply(aN)
```

Summary Problems:

- Loss of function types $\Rightarrow$ Introducing `Fun`N–Interfaces
- `Fun`N–Subtyping problem $\Rightarrow$ Using wildcards
- Impossibility of direct application of lambda expressions
  $\Rightarrow$ Using type-casts

All problems are solvable, but not pretty!!!

Summary Problems:

- Loss of function types $\Rightarrow$ Introducing `Fun`N–Interfaces
- `Fun`N–Subtyping problem $\Rightarrow$ Using wildcards
- Impossibility of direct application of lambda expressions
  $\Rightarrow$ Using type-casts

All problems are solvable, but not pretty!!!

$\Rightarrow$ **Introducing real function types**

Wildcards motivation

```
Vector<? extends Object> v = new Vector<Integer> ();
                           //Subtyping in the parameters is legal

v.addElement(new Object());  //ERROR, as "Object" is not a subtype
                             //of "? extends Object"

Object o = v.elementAt(0);   //legal, as "? extends Object" is a
                             //subtype of "Object"
```

Wildcards motivation

```
Vector<? extends Object> v = new Vector<Integer> ();
                          //Subtyping in the parameters is legal

v.addElement(new Object());  //ERROR, as "Object" is not a subtype
                             //of "? extends Object"

Object o = v.elementAt(0);   //legal, as "? extends Object" is a
                             //subtype of "Object"


Vector<? super Integer> v = new Vector<Object> ();
                         //Supertyping in the parameters is legal

v.addElement(new Integer(1)); //legal, as "Integer" is a
                              //subtype of "? super Integer"

Integer i = v.elementAt(0);  //ERROR, as "? super Integer" is not
                             //a subtype of "Integer"
```

Comparison to function types

There is no motivation for bounded wildcards to allow subtyping in parameters!!!

It holds:

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

which means it should hold:

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

View to Scala

Scala supports variance annotations of type parameters of generic classes. In contrast to Java 5 (aka. JDK 1.5), variance annotations may be added when a class abstraction is defined, whereas in Java 5, variance annotations are given by clients when a class abstraction is used.
[Scala Tutorial]

View to Scala

Scala supports variance annotations of type parameters of generic classes. In contrast to Java 5 (aka. JDK 1.5), variance annotations may be added when a class abstraction is defined, whereas in Java 5, variance annotations are given by clients when a class abstraction is used.
[Scala Tutorial]

```
trait Function_n[-T1 , ... , -Tn, +R] {
  def apply(x1: T1 , ..., xn: Tn): R
  override def toString = "<function>"
}
```

Introduction of Fun$N$*

```
interface Fun$N$*<+R,-T1, ..., -T$N$> {
    R apply(T1 arg1, ..., T$N$ arg$N$);
}
```

where

- Fun$N$*<$T_0, T'_1, \ldots, T'_N$> $\leq^*$ Fun$N$*<$T'_0, T_1, \ldots, T_N$> iff $T_i \leq^* T'_i$
- For Fun$N$* no wildcards are allowed.

Introduction of Fun$N*$

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

where

- $\texttt{Fun}N*\texttt{<}\texttt{T}_0, \texttt{T}'_1, \ldots, \texttt{T}'_N\texttt{>} \leq^* \texttt{Fun}N*\texttt{<}\texttt{T}'_0, \texttt{T}_1, \ldots, \texttt{T}_N\texttt{>}$ iff $\texttt{T}_i \leq^* \texttt{T}'_i$
- For Fun$N*$ no wildcards are allowed.

Proposal: Lambda–expressions are explicitly typed by FunN*–types

Solved Problems

- Lambda–expressions has types
- `Fun`$N*$–types allows subtyping without wildcards
- Direct application of lambda–expressions is possible without type-casts, as lambda–expressions have types.

Function types and type-erasures

Overloading example

```
void apply(Fun*1<Integer, Integer> f) { ... }

void apply(Fun*1<Boolean, Boolean> f) { ... }
```

Function types and type-erasures

Overloading example

```
void apply(Fun*1<Integer, Integer> f) { ... }

void apply(Fun*1<Boolean, Boolean> f) { ... }
```

Leads in byte-code to (type-erasure):

```
void apply(Fun*1 f) { ... }

void apply(Fun*1 f) { ... }
```

Ambigous overloading!

Translation without Type-Erasures

- *Generic instances in Java Byte Code* [Pluemicke 2014, Bad Honnef]

  Approach to solve class loading performance problem

  - Idea: Instantiated types are subtypes of the non-instantiated type [Ureche, Talau, Odersky: *Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations*, OOPSLA, 2013]

Translation without Type-Erasures

- *Generic instances in Java Byte Code* [Pluemicke 2014, Bad Honnef]

  Approach to solve class loading performance problem

  - Idea: Instantiated types are subtypes of the non-instantiated type [Ureche, Talau, Odersky: *Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations*, OOPSLA, 2013]

- *Two Ways to Bake Your Pizza - Translating Parameterised Types into Java* [Odersky, Runne, Wadler 2000]

  Approach to solve class loading performance problem

  - changed class loader, that needs only loading a part of the type instanced classes.

Integration Functional Interfaces and `Fun`$N*$–Type

- Java lambda–expressions get `Fun`$N*$-Types as explicite types.
- Target typing by functional interfaces are preserved.
- A target type is compatible, if its method's type is a supertype of the `Fun`$N*$–type.

Brian Goetz's (Oracle) arguments against real function types

- 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)`.

Brian Goetz's (Oracle) arguments against real function types

- 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)`.

Brian Goetz's (Oracle) arguments against real function types

- 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)`.

Brian Goetz's (Oracle) arguments against real function types

- 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)`.

Conclusion and Outlook

Conclusion

- Problems with functional interfaces as target types of lambda–expressions in Java 8.
- Simulating of function types by functional interfaces `FunN`.
- Introduction of Scala-like functional interfaces into Java.
- Integration of both approaches.

Conclusion and Outlook

Conclusion

- Problems with functional interfaces as target types of lambda–expressions in Java 8.
- Simulating of function types by functional interfaces `FunN`.
- Introduction of Scala-like functional interfaces into Java.
- Integration of both approaches.

Outlook

- Theoretical foundation
- Implementation
- Bytecode without type-erasures

Stellenausschreibung

DHBW (M. Plümicke) gemeinsam mit der Uni Freiburg (P. Thiemann)

Wissenschaftlicher Mitarbeiter/in

Promotionsthema: Real function types in Java

Bei Interesse: Martin Plümicke, pl@dhbw.de, 07451-521142