Functional Interfaces vs. Function Types in Java with Lambdas

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

26. February 2014

Overview

Two approaches

Java 8: [Goetz 2013: State of the Lambda]:

- Lambdas with functional interfaces as target types

Java$_\lambda$: [Project Lambda 2010, version 0.1.5; Plümicke, PPPJ 2011]:

- Lambdas with real function-types

Arguments for functional interfaces [Goetz 2013]

- Mixture of structural and nominal types
- Divergence of library styles:
  - some libraries would continue to use callback interfaces
  - others would use structural function types
- Unwieldy syntax
- Erasures limit overloading of generic function types

Lambda–expressions in Java 8

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

Lambda–expressions in Java 8

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

void doAddition(Operation o) { ... }  //functional interface
```

Lambda–expressions in Java 8

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

void doAddition(Operation o) { ... }  //functional interface

doAddition(new Operation () {          // anonymous inner class
                 public int op (int x, int y) {
                     return x + y;
                 }
              });
```

Lambda–expressions in Java 8

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

void doAddition(Operation o) { ... }  //functional interface

doAddition(new Operation () {          // anonymous inner class
                public int op (int x, int y) {
                    return x + y;
                }
             });

doAddition((int x, int y) -> x + y)   // lambda expressions
```

Lambda–expressions in Java$_\lambda$

```
void doAddition(#int(int, int) o) { ... } // function type
```

```
doAddition((int x, int y) -> x + y)    // lambda expressions
```

Function–types

For $ty, ty_i \in \text{Type}_{TS}(\, BTV \,)$

$$\# \, ty \, (ty_1, \ldots, ty_n) \in \text{Type}_{TS}(\, BTV \,)$$ [1]

[1] Sometimes function types $\# \, ty \, (ty_1, \ldots, ty_n)$ are written as $(ty_1, \ldots, ty_n) \; \rightarrow \; ty$.

Function–types

For $ty, ty_i \in \mathrm{Type}_{TS}(\,BTV\,)$

$$\#\, ty\, (ty_1, \ldots, ty_n) \in \mathrm{Type}_{TS}(\,BTV\,)$$ [1]

Subtyping:

$$\#\,\theta\,(\theta'_1, \ldots, \theta'_n) \leq^* \#\,\theta'\,(\theta_1, \ldots, \theta_n) \quad \textit{iff} \quad \theta \leq^* \theta' \textit{ and } \theta_i \leq^* \theta'_i.$$

[1] Sometimes function types $\#\, ty\, (ty_1, \ldots, ty_n)$ are written as $(ty_1, \ldots, ty_n) \;\rightarrow\; ty$.

Matrix example in Java$_\lambda$ with function–types

```
class Matrix extends Vector<Vector<Integer>> {
  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = (m) -> (f) -> f.(Matrix.this, m);
```

Matrix example in Java$_\lambda$ with function–types

```
class Matrix extends Vector<Vector<Integer>> {
  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = (m) -> (f) -> f.(Matrix.this, m);

  //(Vector<Vector<Integer>>, Vector<Vector<Integer>>) -> Matrix
  #Matrix(Vector<Vector<Integer>>, Vector<Vector<Integer>>)
  mul = (m1,m2) -> { ... }
```

Matrix example in Java$_\lambda$ with function–types

```
class Matrix extends Vector<Vector<Integer>> {
  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = (m) -> (f) -> f.(Matrix.this, m);

  //(Vector<Vector<Integer>>, Vector<Vector<Integer>>) -> Matrix
  #Matrix(Vector<Vector<Integer>>, Vector<Vector<Integer>>)
  mul = (m1,m2) -> { ... }

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

Matrix example in Java$_\lambda$ with function–types

```
class Matrix extends Vector<Vector<Integer>> {
  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = (m) -> (f) -> f.(Matrix.this, m);

  //(Vector<Vector<Integer>>, Vector<Vector<Integer>>) -> Matrix
  #Matrix(Vector<Vector<Integer>>, Vector<Vector<Integer>>)
  mul = (m1,m2) -> { ... }

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

#Matrix(Vector<Vector<Integer>>,Vector<Vector<Integer>>)

$\leq^*$ # Matrix (Matrix, Matrix)

Types of lambda expressions in Java 8

Functional interfaces: Interfaces with a single method (SAM–types) as target types for lambda expressions.

Types of lambda expressions in Java 8

Functional interfaces: Interfaces with a single method (SAM–types) as target types for lambda expressions.

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(); }
```

Target typing

- The lambda expressions have no explicit types
- The target types are deduced from the context
- Not all target types are correct

Target typing

- The lambda expressions have no explicit types
- The target types are deduced from the context
- Not all target types are correct

```
Operation O = (int x, int y) -> x + y;    //correct
Operation O = (String s) -> s + s;        //incorrect
```

Target typing

- The lambda expressions have no explicit types
- The target types are deduced from the context
- Not all target types are correct

```
Operation O = (int x, int y) -> x + y;    //correct
Operation O = (String s) -> s + s;        //incorrect
```

A correct target type is called compatible.

Functional interfaces as compatible target types

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

Functional interfaces as compatible target types

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

Lemma: There is an equivalence class of compatible target types for a lambda expression.

Canonical representative

For the equivalence class of the compatible target types of a lambda expression, there is a canonical representative

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

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

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

Example (compatible)

```
interface Fun1<R,T> { R apply(T arg); }

interface Add { Fun1<Integer,Integer> add (Integer a); }
```

Example (compatible)

```
interface Fun1<R,T> { R apply(T arg); }

interface Add { Fun1<Integer,Integer> add (Integer a); }
```

- `(Integer x) -> (Integer y) -> (x + y)` is compatible with `Add`
- `(Integer y) -> (x + y)` is compatible with `Fun1<Integer, Integer>`

Example matrix with functional interfaces

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

  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
  op = (m) -> (f) -> f.apply(this, m);
```

Example matrix with functional interfaces

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

  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
  op = (m) -> (f) -> f.apply(this, m);

  //(Matrix, Matrix) -> Matrix
  Fun2<Matrix, Matrix,Matrix>
  mul = (m1, m2) -> { ... }
```

Example matrix with functional interfaces

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

  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
  op = (m) -> (f) -> f.apply(this, m);

 //(Matrix, Matrix) -> Matrix
  Fun2<Matrix, Matrix,Matrix>
  mul = (m1, m2) -> { ... }

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

Example matrix with functional interfaces

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

  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
  op = (m) -> (f) -> f.apply(this, m);

  //(Matrix, Matrix) -> Matrix
  Fun2<Matrix, Matrix,Matrix>
  mul = (m1, m2) -> { ... }

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

Fun2<Matrix, Vector<Vector<Integer>>,>Vector<Vector<Integer>>.
$\not\leq^*$ Fun2<Matrix, Matrix, Matrix>

Subtyping

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

Subtyping

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

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

Subtyping

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

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

Background: Variance of type parameters

Example (result types):

$$\texttt{Integer} \rightarrow \texttt{Integer} \quad \leq^* \quad \texttt{Integer} \rightarrow \texttt{Object}$$

Example (result types):

$$\texttt{Integer} \rightarrow \texttt{Integer} \quad \leq^* \quad \texttt{Integer} \rightarrow \texttt{Object}$$

but

```
Fun1<Integer,Integer> idIntInt = (Integer x) -> x
Fun1<Object,Integer> idIntObj = idIntInt
```

is **wrong!**, as

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

Example (result types):

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

but

```
Fun1<Integer,Integer> idIntInt = (Integer x) -> x
Fun1<Object,Integer> idIntObj = idIntInt
```

is **wrong!**, as

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

```
Fun1<Integer,Integer> idIntInt = (Integer x) -> x;
Fun1<? extends Object, Integer> idIntExtObj = idIntInt;
```

is **correct!**, as

$$\texttt{Fun1<Integer,Integer>} \quad \leq^* \quad \texttt{Fun1<? extends Object, Integer>}$$

Example (argument type)

$$\texttt{Number} \rightarrow \texttt{Integer} \quad \leq^* \quad \texttt{Integer} \rightarrow \texttt{Integer}$$

Example (argument type)

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

but

```
Fun1<Integer,Number> idNumInt = (Number x) -> (Integer)x
Fun1<Integer,Integer> idIntInt = idNumInt
```

is **wrong!**, as

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

Example (argument type)

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

but

```
Fun1<Integer,Number> idNumInt = (Number x) -> (Integer)x
Fun1<Integer,Integer> idIntInt = idNumInt
```

is **wrong!**, as

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

```
Fun1<Integer, Number> idNumInt = (Number x) -> (Integer)x;
Fun1<Integer, ? super Integer> idSupIntInt = idNumInt;
```

is **correct!**, as

$$\texttt{Fun1<Integer, Number>} \leq^* \texttt{Fun1<Integer, ? super Integer>}$$

Example (combination argument and result type)

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

Example (combination argument and result type)

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

```
Fun1<Integer, Number> idNumInt = (Number x) -> (Integer)x;
Fun1<? extends Object, ? super Integer> idSupIntExtObj = idNumInt;
```

Example (combination argument and result type)

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

```
Fun1<Integer, Number> idNumInt = (Number x) -> (Integer)x;
Fun1<? extends Object, ? super Integer> idSupIntExtObj = idNumInt;
```

as

$$\texttt{Fun1<Integer, Number>} \leq^* \texttt{Fun1<? extends Object, ? super Integer>}$$

Type parameter subtyping in Java 8

It holds

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

for $T_i \leq^* T'_i$.

Example matrix with functional interfaces and subtyping

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

  Fun1<Fun1<Mat, Fun2<Mat, ? super Mat, ? super Mat>>, Mat>
  op = (m) -> (f) -> f.apply(this, m);
```

Example matrix with functional interfaces and subtyping

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

  Fun1<Fun1<Mat, Fun2<Mat, ? super Mat, ? super Mat>>, Mat>
  op = (m) -> (f) -> f.apply(this, m);

  Fun2<Mat, Vector<Vector<Integer>>,Vector<Vector<Integer>>>
  mul = (m1, m2) -> { . . . }
```

Example matrix with functional interfaces and subtyping

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

  Fun1<Fun1<Mat, Fun2<Mat, ? super Mat, ? super Mat>>, Mat>
  op = (m) -> (f) -> f.apply(this, m);

  Fun2<Mat, Vector<Vector<Integer>>,Vector<Vector<Integer>>>
  mul = (m1, m2) -> { . . . }

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

Example matrix with functional interfaces and subtyping

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

  Fun1<Fun1<Mat, Fun2<Mat, ? super Mat, ? super Mat>>, Mat>
  op = (m) -> (f) -> f.apply(this, m);

  Fun2<Mat, Vector<Vector<Integer>>,Vector<Vector<Integer>>>
  mul = (m1, m2) -> { . . . }

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

Fun2<Mat, Vector<Vector<Integer>>,>Vector<Vector<Integer>>.

$\leq^*$ Fun2<Mat, ? super Mat, ? super Mat>

Function-application

in Java$_\lambda$ (with type–inference):

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

Function-application

in Java$_\lambda$ (with type–inference):

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

in Java 8:

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

wrong!, no lambda expression is allowed as receiver

Function-application

in Java$_\lambda$ (with type–inference):

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

in Java 8:

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

wrong!, no lambda expression is allowed as receiver

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

ok!, as there is cast–expression

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

Application:

in Java 8:

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

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

Application:

in Java 8:

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

in Java$_\lambda$: (with type–inference):

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

Simulation of function-types by functional interfaces

Summary:

1. Introduction of canonical representatives `Fun`*N*
2. Subtyping by using wildcards
3. Currying by introduction of type-casts

Simulation of function-types by functional interfaces

Summary:

1. Introduction of canonical representatives `Fun`N
2. Subtyping by using wildcards
3. Currying by introduction of type-casts

Possible but not beautiful!

Proposal: functional interfaces and function-types

- Lambda expressions as in Java 8
- Function-types as types of lambda expressions, as in Java$_\lambda$
- Functional interfaces as target types as in Java 8
- An evaluation-operator that allows to apply a lambda expression to its arguments, directly.

Discussion

The approach leads to solutions for two Goetz's problems:

Discussion

The approach leads to solutions for two Goetz's problems:

Mixture of structural and nominal types:

Each structural type $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ is equivalent to an nominal type $\texttt{Fun}n\texttt{<}\tau, \tau_1, \ldots, \tau_n\texttt{>}$.

Divergence of library style:

Furthermore functional interfaces can be used as library arguments.

Discussion

The approach leads to solutions for two Goetz's problems:

Mixture of structural and nominal types:
Each structural type $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ is equivalent to an nominal type $\texttt{Fun}n\texttt{<}\tau, \tau_1, \ldots, \tau_n\texttt{>}$.

Divergence of library style:
Furthermore functional interfaces can be used as library arguments.

The others problems:

Unwieldy syntax: Since the introduction of wildcards the syntax is already unwieldy.
It can be solved by introduction of complete type inference.

Erasures limit overloading of generic function types: This problem is also given by using functional interfaces $\texttt{Fun}N\texttt{<}a, a_1, \ldots, a_N\texttt{>}$

Lambda expressions in other object-oriented languages

Scala:

Subtyping of function-types: OK (declaration of type parameters as *contravariant*, *covariant*)

Direct evaluation of lambda expressions: OK (`apply`-method)

Lambda expressions in other object-oriented languages

Scala:

- Subtyping of function-types: OK (declaration of type parameters as *contravariant*, *covariant*)
- Direct evaluation of lambda expressions: OK (`apply`-method)

C#:

- Subtyping of function-types: OK (declaration of type parameters as *contravariant*, *covariant*)
- Direct evaluation of lambda expressions: no

Lambda expressions in other object-oriented languages

Scala:

- Subtyping of function-types: OK (declaration of type parameters as *contravariant*, *covariant*)
- Direct evaluation of lambda expressions: OK (`apply`-method)

C#:

- Subtyping of function-types: OK (declaration of type parameters as *contravariant*, *covariant*)
- Direct evaluation of lambda expressions: no

C++:

- Subtyping of function-types: no
- Direct evaluation of lambda expressions: OK

Conclusion and Outlook

Conclusion

- Lambda expressions in Java 8
- Equivalence class of the compatible target types of a lambda expression and a canonical representative
- Function-types vs. functional interfaces
  - Subtyping
  - Direct evaluation of lambda expressions
- Proposal: Function-types and functional interfaces in Java

Conclusion and Outlook

Conclusion

- Lambda expressions in Java 8
- Equivalence class of the compatible target types of a lambda expression and a canonical representative
- Function-types vs. functional interfaces
  - Subtyping
  - Direct evaluation of lambda expressions
- Proposal: Function-types and functional interfaces in Java

Outlook

- Implementation of the Proposal
- Consideration of overloading of generic function types