Functional Interfaces vs. Function Types in Java with Lambdas

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

26. February 2014
Overview

Introduction

Functional interfaces vs. function-types
  Function-types
  Functional interfaces
  Subtyping
  Evaluation of lambda expressions

Proposal: functional interfaces and function-types

Conclusion and Outlook
Two approaches

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

▷ Lambdas with functional interfaces as target types

Java₇: [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

```java
interface Operation {
    public int op (int x, int y);
}
```
Lambda–expressions in Java 8

```java
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

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

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

```java
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 \( ty, ty_i \in \text{Type}_{TS}(BTV) \)

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

Subtyping:

\[
\# \theta\left(\theta'_1, \ldots, \theta'_n\right) \leq^* \# \theta'\left(\theta_1, \ldots, \theta_n\right) \quad \text{iff} \quad \theta \leq^* \theta' \text{ and } \theta_i \leq^* \theta'_i.
\]

\(^1\text{Sometimes function types } \# ty\left(ty_1, \ldots, ty_n\right) \text{ are written as } (ty_1, \ldots, ty_n) \rightarrow ty.\)
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>>) -> Matrix
  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\ast #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.

```java
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 0 = (int x, int y) → x + y; //correct
Operation 0 = (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
A lambda expression is \textit{compatible} with a type $T$, if

- $T$ is a functional interface type
- The lambda expression has the \textit{same number} of parameters as $T$’s method, and those parameters’ types are the same
- Each expression returned by the lambda body is \textit{compatible} with $T$’s method’s return type
- Each exception thrown by the lambda body is \textit{allowed by} $T$’s method’s throws clause

\textbf{Lemma:} There is an \textit{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

$$\text{Fun}N\langle R, T_1, \ldots, T_N \rangle$$

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

$$(T_1, \ldots, T_N) \rightarrow R$$
Example (compatible)

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

interface Add { Fun1<Integer,Integer> add (Integer a); }
```
interface Fun1<R,T> { R apply(T arg); }

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

- (Integer x) \rightarrow (Integer y) \rightarrow (x + y) is compatible with Add
- (Integer y) \rightarrow (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>>.

\* Fun2<Matrix, Matrix, Matrix>
Subtyping

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

\[(T_1', \ldots, T_N') \rightarrow T_0 \leq^* (T_1, \ldots, T_N) \rightarrow T_0', \quad \text{iff} \quad T_i \leq^* T'_i\]

\[\text{Fun}_N<T_0, T_1', \ldots, T_N'> \not\leq^* \text{Fun}_N<T_0', T_1, \ldots, T_N'>, \quad \text{for} \quad T_i \not\leq^* T'_i\]
Subtyping

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

\[\text{FunN}<T_0, T'_1, \ldots, T'_N> \not\leq^* \text{FunN}<T'_0, T_1, \ldots, T_N>, \quad \text{for} \quad T_i \not\leq^* T'_i\]

**Background:** Variance of type parameters
Example (result types):

\[
\text{Integer} \rightarrow \text{Integer} \leq^* \text{Integer} \rightarrow \text{Object}
\]
Example (result types):

\[
\text{Integer} \rightarrow \text{Integer} \quad \leq^* \quad \text{Integer} \rightarrow \text{Object}
\]

but

\[
\text{Fun1<Integer,Integer>} \ id\text{IntInt} = (\text{Integer } x) \rightarrow x
\]

\[
\text{Fun1<Object,Integer>} \ id\text{IntObj} = id\text{IntInt}
\]

is **wrong!**, as

\[
\text{Fun1<Integer,Integer>} \nleq^* \text{Fun1<Object, Integer>}
\]
Example (result types):

$$\text{Integer} \rightarrow \text{Integer} \preceq^* \text{Integer} \rightarrow \text{Object}$$

but

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

is **wrong**!, as

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

is **correct**!, as

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

\[
\text{Number} \rightarrow \text{Integer} \preceq^* \text{Integer} \rightarrow \text{Integer}
\]
Example (argument type)

\[
\text{Number} \rightarrow \text{Integer} \; \leq^* \; \text{Integer} \rightarrow \text{Integer}
\]

but

\[
\text{Fun1}\langle\text{Integer}, \text{Number}\rangle \; \text{idNumInt} = \text{(Number x)} \rightarrow \text{(Integer)x}
\]
\[
\text{Fun1}\langle\text{Integer}, \text{Integer}\rangle \; \text{idIntInt} = \text{idNumInt}
\]

is wrong!, as

\[
\text{Fun1}\langle\text{Integer}, \text{Number}\rangle \; \not\leq^* \; \text{Fun1}\langle\text{Integer}, \text{Integer}\rangle
\]
Example (argument type)

\[ \text{Number} \rightarrow \text{Integer} \quad \leq^* \quad \text{Integer} \rightarrow \text{Integer} \]

but

\[
\begin{align*}
\text{Fun1<Integer,Number>} \ &\text{idNumInt} = (\text{Number} \ x) \rightarrow (\text{Integer})x \\
\text{Fun1<Integer,Integer>} \ &\text{idIntInt} = \text{idNumInt}
\end{align*}
\]

is wrong!, as

\[
\begin{align*}
\text{Fun1<Integer,Number>} \ &\not\leq^* \quad \text{Fun1<Integer,Integer>}
\end{align*}
\]

\[
\begin{align*}
\text{Fun1<Integer, Number>} \ &\text{idNumInt} = (\text{Number} \ x) \rightarrow (\text{Integer})x; \\
\text{Fun1<Integer, ? super Integer>} \ &\text{idSupIntInt} = \text{idNumInt};
\end{align*}
\]

is correct!, as

\[
\begin{align*}
\text{Fun1<Integer, Number>} \ &\leq^* \text{Fun1<Integer, ? super Integer>}
\end{align*}
\]
Example (combination argument and result type)

\[
\text{Number} \rightarrow \text{Integer} \leq^* \text{Integer} \rightarrow \text{Object}
\]
Example (combination argument and result type)

\[
\text{Number} \rightarrow \text{Integer} \leq \ast \quad \text{Integer} \rightarrow \text{Object}
\]

```java
Fun1<Integer, Number> idNumInt = (Number x) -> (Integer)x;
Fun1<? extends Object, ? super Integer> idSupIntExtObj = idNumInt;
```
Example (combination argument and result type)

\[ \text{Number} \rightarrow \text{Integer} \leq^* \text{Integer} \rightarrow \text{Object} \]

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

\text{as}

Fun1<Integer, Number> \leq^* \text{Fun1<? extends Object, ? super Integer>}
```
Type parameter subtyping in Java 8

It holds

$$
\begin{align*}
\text{Fun}_N<T_0, T_1', \ldots, T_N'> & \leq^* \\
\text{Fun}_N<\text{? extends } T_0', \text{? super } T_1, \ldots, \text{? super } T_N'>, \\
\text{for } T_i & \leq^* T_i'.
\end{align*}
$$
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);
}

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>>>

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);
    }

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>>>
    .≤
    Fun2<Mat, ? super Mat, ? super Mat>
    ∗

    Martin Plümicke

Function-application

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

\[
((x_1,\ldots, x_N) \to h(x_1,\ldots, x_N)).(a_1,\ldots,a_N);
\]
Function-application

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

\[((x_1,\ldots, x_N) \rightarrow h(x_1,\ldots, x_N)).(\text{arg}1\ldots,\text{arg}N)\];

in Java 8:

\[((x_1,\ldots, x_N) \rightarrow h(x_1,\ldots, x_N)).\text{apply}(\text{arg}1\ldots,\text{arg}N)\];

wrong!, no lambda expression is allowed as receiver
Function-application

in Java_\lambda_ (with type–inference):

\((x_1,\ldots, x_N) \rightarrow h(x_1,\ldots, x_N))\cdot\text{apply}(\text{arg1},\ldots,\text{argN});\)

wrong!, no lambda expression is allowed as receiver

\(((\text{FunN}<\text{T}_0, \text{T}_1,\ldots, \text{T}_N>)\)
\((x_1,\ldots, x_N) \rightarrow h(x_1,\ldots, x_N))\cdot\text{apply}(\text{arg1},\ldots,\text{argN});\)

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:

```java
((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:

$((Fun1<Fun1<Fun1<\ldots Fun1<T0, TN>, \ldots>, T2>, T1> )$

$(x1) \rightarrow (x2) \rightarrow \ldots \rightarrow (xN) \rightarrow h(x1, \ldots ,XN))$

.apply(a1).apply(a2)....apply(aN))

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

$((x1) \rightarrow (x2) \rightarrow \ldots \rightarrow (xN) \rightarrow h(x1, \ldots ,XN)) .a1 .a2 \ldots .aN$
Simulation of function-types by functional interfaces

Summary:

1. Introduction of canonical representatives $\text{Fun}N$
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 $\text{FunN}$
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:

Mixture of structural and nominal types: Each structural type \((\tau_1, \ldots, \tau_n)\) is equivalent to a nominal type \(\text{Fun}_n<\tau, \tau_1, \ldots, \tau_n>\).

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

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 \(\text{Fun}_N<a, a_1, \ldots, a_N>\).
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 \(\text{Fun}_n<\tau, \tau_1, \ldots, \tau_n>\).

**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 \(\text{Fun} n<\tau, \tau_1, \ldots, \tau_n>\).

**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 \(\text{Fun} N<a, a_1, \ldots, a_N>\).
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 \textit{contravariant, covariant})
- Direct evaluation of lambda expressions: OK (\texttt{apply}-method)

C#:
- Subtyping of function-types: OK (declaration of type parameters as \textit{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 \textit{contravariant}, \textit{covariant})
Direct evaluation of lambda expressions: OK (\texttt{apply}-method)

C#:

Subtyping of function-types: OK (declaration of type parameters as \textit{contravariant}, \textit{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