Brian's approach vs. Martin's approach Functional Interfaces vs. function-types in Java 8

Martin Plümicke

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

Abstract. In the Project lambda[1] a new version (version 8) of Java is developed (Brian's approach). The java language architect is Brian Goetz.
According to Martin's approach, the langugage Java_λ was derived from an earlier version of the language.
While in Java 8 functional interfaces (interfaces with one method) are target types of lambda expressions, in Java_λ real function-types are lambda expression types. This induces that subtyping of lambda expressions types is completely different in Java 8 and Java_λ.
In Java 8 there is no apply/eval–operator which applies a lambda expression to arguments. In Java 8 the application of a lambda expression can only be done by a method call of the corresponding method in the functional interface.
Futhermore we consider type–inference in Java 8 and in Java_λ.

1 Introduction

The most important goal of the new Java version is to introduce programming patterns that allow modeling code such as data [Goe11]. Brian's approach includes the new features closures, functional interfaces as target types, method and constructor references and default methods.
Our, the Martin's approach, bases on an earlier version of Java 8 [Goe10]. We call a core of the language Java_λ, which we presented in [Plü11]. It includes the new features closures, **function-types**, **higher-order functions** and **complete type-inference**.
With the example which is also described in [Goe11], we want to show the extensions of Java 8. The task of the example is sorting a list of people by last name. As of today we write:

```
Collections.sort(people, new Comparator<Person>() {
  public int compare(Person x, Person y) {
    return x.getLastName().compareTo(y.getLastName());
  }});
```

[1] (http://openjdk.java.net/projects/lambda)

With lambda expressions we can avoid the inelegant inline instantiation of the implementing class:

```
Collections.sort(people,
    (Person x, Person y) ->
        x.getLastName().compareTo(y.getLastName()));
```

The type inference mechanism of Java 8 allows to omit the argument types:

```
Collections.sort(people,
    (x, y) ->
        x.getLastName().compareTo(y.getLastName()));
```

In Java 8 such types of lambda expressions are called compatible target types. A target type of a lambda expressions is a functional interface[2].

Furthermore, we can simplify this example by introducing the method `comparing` whereupon `comparing` takes a function for mapping each value to a sort key and returns an appropriate comparator.

```
public <T, U extends Comparable<? super U>>
  Comparator<T> comparing(Mapper<T, ? extends U> mapper)
  { ... }

interface Mapper<T,U> { public U map(T t); }

Collections.sort(people, comparing(p -> p.getLastName()));
```

The above mentioned lambda expression is only a forwarder to the method `getLastName`. We can use the Java 8 feature method references to reuse the existing method in place of the lambda expression:

```
Collections.sort(people, comparing(Person::getLastName));
```

Method reference in Java 8 means referring to a method (or a constructor) of an existing class or object whose typing is compatible with the corresponding functional type.

2 Functional interfaces vs. function-types

In this section we will present the target types of closures, which are given as functional interfaces in Brian's approach. We will compare the target types with the function-types, which are in given in Martin's approach. We consider the different definitions, the subtyping property and higher-order functions.
The set of Java types Ty_p in a Java 8 program p is given as in [GJSB05], Section 4.5 and in [Plü07c], Section 2[3].

[2] In earlier publications (e.g. [lam10]) functional interfaces are called SAM-types.
[3] In [Plü07c], [Plü11] we called the Java types simple types $\mathsf{SType}_{TS}(\,BTV\,)$.

2.1 Functional interfaces as compatible target types

For the type descriptions of methods the Java types are not expressive enough. Therefore function-types are defined:

Definition 1 (Function-types). *Let Ty_p be a set of Java types. The set of function-types FTy_p is defined by*

- $Ty_p \subseteq Fty_p$
- *For* $ty, ty_i \in Fty_p : ty_1 \times \ldots \times ty_n \to ty \in Fty_p$

In Brian's approach function types are only used for the description of methods. They are not used in Java 8 programs.

Definition 2 (Functional interface). *An interface I, which has only one method, is called a functional interface.*

Example 1. Many common callback interfaces have this property:

```
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(); }
interface PrivilegedAction<T> { T call(); }
```

Definition 3 (Compatible). *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*[4]

The compatible condition we denote by $Comp(lexp, T)$.

Example 2. Let the functional interfaces `Fun1` and `Add` be given:

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

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

The lambda expression

[4] Exceptions are not considered in this paper.

```
(Integer x) ->  (Integer y) -> (x + y)
```

is compatible with `Add`, as the type of the argument `a` respectively `x` is `Integer` and `(Integer y) -> (x + y)` is compatible with `Fun1<Integer,Integer>` .

Example 3. The same lambda expressions can be compatible with different target types. E.g. the lambda expression

```
() -> "done";
```

is compatible with target types `Callable<String>` and `PrivilegedAction<String>` (e.g. Example 1).

Remark 1. The notion of compatible types is continued analogously on method and constructor references. Additionally the condition $Comp(\, m : ty, T\,)$ is defined analogously for method and constructor references.

We give an additional definition, which defines two functional interfaces as equivalent, if their methods are equal.

Definition 4 (Equivalent functional interfaces). *Two functional interfaces are equivalent (in sign: $\sim_{fi}$) if for its single methods holds:*

- *the number of arguments and its types are equal*
- *the result types are equal or equivalent*

Lemma 1. *The relation $\sim_{fi}$ is an equivalence relation.*

Example 4. In Example 3 `Callable<T>` and `PrivilegedAction<T>` are equivalent.

Additionally we will define a canonical functional interface for each equivalence class. Therefore we consider again the interface

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

from Example 2. This interface stands for a functional interface with a method with one argument. For each functional interface with one argument there is an instance of `Fun1`, which is equivalent. This instance can be considered as a canonical representation. In the following we generalize this idea. We extend Java 8 by the following set of interfaces.

Definition 5 (Interface `Fun`N). *The language Java 8 is extended for all $N \in \mathbb{N}$ by*

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

This leads directly to the following theorem.

Theorem 1 (Canonical representation). *For each functional interface there is an unique N, such that an instance of* `FunN` *is an equivalent functional interface.*
This instance is called canonical representation of the equivalence class of functional interfaces.

Example 5. The canonical representation of the compatible types of the lambda expression `() -> "done"` from Example 3 is `Fun0<String>`.

According to Martin's approach, function-types (Def. 1) are given as types of lambda expression.

Example 6. The lambda–expression

```
(Person x, Person y) -> x.getLastName().compareTo(y.getLastName())
```

has the type:

$$(\texttt{Person}, \texttt{Person}) \to \texttt{int}$$

Remark 2. The canonical representation `FunN<R,T1 , ..., TN>` in Brian's approach can play a similar role as the function-types in Martin's approach.
In the next sections we will explain the differences.

2.2 Subtyping

For function-types holds

$$T_1' \times \ldots \times T_N' \to T_0 \;\leq^*\; T_1 \times \ldots \times T_N \to T_0', \quad \text{iff } T_i \leq^* T_i'.$$

But for the canonical representation holds

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

Otherwise the Java type system would not be sound (cp. [Plü07a]).
Let us consider the following example

Example 7. It holds

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

but

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

is **wrong!**, as

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

Therefore in the Java type system wildcards are introduced.
It holds

$$A\texttt{<}T\texttt{>} \leq^* A\texttt{<? extends }T'\texttt{>}, \text{ iff } T \leq^* T'$$

and

$$A\texttt{<}T'\texttt{>} \leq^* A\texttt{<? super }T\texttt{>}, \text{ iff } T \leq^* T'.$$

With this feature we consider the example again.

Example 8. It holds

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

as `Fun1<Integer,Integer>` $\leq^*$ `Fun1<? extends Object, Integer>` and

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

as `Fun1<Integer, Number>` $\leq^*$ `Fun1<? extends Object, ? super Integer>`

Finally we present a function, which takes an integer and a function. The result is the application of the function to the integer.

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

Fun1<Integer,Integer> idIntInt = (Integer x) -> (Integer)x;
Object x1 = m(2, idIntInt);

Fun1<Object,Integer> idIntObj = (Integer x) -> (Object)x;
Object x2 = m(2, idIntObj);

Fun1<Integer, Number> idNumInt = (Number x) -> (Integer)x;
Object x3 = m(2, idNumInt);
```

The method shows that all subtypes of the result types and all supertypes of the argument types are correct, if we use wildcards.

We generalize this result.

Theorem 2. *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 $T_i \leq^* T'_i$.

This result is comparable to the property of function-types.

2.3 Higher-order functions

A higher-order function is function, where the arguments or the results are also functions. In the λ–calculus, which is the theoretical base of all languages with closures, lambda expressions can be applied directly to their arguments like:

$$((x1, \ldots, xN) \rightarrow h(x1, \ldots, xN))(arg1 \ldots, argN).$$

In Martin's approach in Java$_\lambda$ such applications are allowed:

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

Therefore the type–inference in Java$_\lambda$ is extended such that the types of lambda expressions are inferred. Furthermore following [lam10] the applications operator dot is introduced.
Following the ideas of functional interfaces in Brian's approach the dot is replaced by the name of the interface's method:

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

Expressions such as explained are not allowed. No lambda expressions are allowed as receivers for method applications. The reason is the ambiguity of the compatible target type of a lambda expression. Therefore a type-cast has to be introduced:

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

This means for the application of a curried function $f : T_1 \rightarrow \ldots \rightarrow T_N \rightarrow T_0$:

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

In contrast in Java$_\lambda$ with type–inference it is possible to write:

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

3 Type–inference

As seen above in Java some types are inferred. Additionally type-inference can simplify the expressions as shown by the lambda expression application. In this section we will show which types are inferred in Java 8, Java$_\lambda$ and the extended Java 8.

The following three kinds of type–inference are introduced in Java 8:

- Type parameter instantiation (Java 5.0):
  `id: a` $\rightarrow$ `a`
  `id(1)` : for `a` the type `Integer` is inferred.

– Diamond–operator (Java 7):

```
Vector <Integer> v = new Vector <>
```

The type can be omitted by the instance creation.

– Parameter's type–inference in lambda expressions (Java 8):

```
(T1 x1, ...,  TN xN) → h( x1,...,XN )
```

The types `T1, ..., TN` can be inferred:

```
(x1, ..., xN) → h( x1, ..., XN )
```

is a correct lambda expression in Java 8.

In contrast in Java_λ

– type-inference of parameter's type in lambda expressions, as in Java 8
– type-inference of complete lambda expressions
– type-inference of local variables

are given. The results of the type-inference algorithm are well-typings [FM88]. A well-typing is a conditional type for an expression, where the conditions are given by a set of consistent constraints.

Example 9. The emphasized types are inferred. It it possible to omit these type declarations.

```
class Example {

  #Integer(Integer) fac = (Integer n) -> {
    Integer ret = 1;
    for (Integer i=2; i <= n; i++) ret = ret * i;
    return ret;
  }}
```

We gave a type-inference system for Java 8 [Plü12], which bases on our type unification [Plü07b]. We extended Java_λ type-inference for methods and replaced the function-types by functional interfaces. The main challenge is overloading and overriding. The results of the type-inference algorithm are therefore sets of types with some constraints.

4 Conclusion and Outlook

We presented the main extensions of Java 8 and compared the approaches of Brian and Martin: functional interfaces vs. function-types. We showed the different consequences with respect to subtyping and higher-order functions. Finally we presented type-inference in Java 8 and our extensions.
In future it should be discussed again if the approach of functional interfaces as compatible target types for lambda expressions is a good decision or if the function-types extended by type-inference would be an alternative. If Brian's approach with functional interfaces would maintained, the standard functional interfaces `Fun`N could be introduced and subtyping could be simplified.

References

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. Proceedings 2nd European Symposium on Programming (ESOP '88), pages 94–114, 1988.

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

[Goe10] Brian Goetz. State of the lambda. 10 October 2010.

[Goe11] Brian Goetz. State of the lambda. December 2011.

[lam10] Project lambda: Java language specification draft. 2010. Version 0.1.5.

[Plü07a] Martin Plümicke. Formalization of the Java Type System. In Michael Hanus and Bernd Braßel, editors, Programmiersprachen und Rechenkonzepte: 24. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte", Bad Honnef, 2. - 4. Mai 2007, number 0707 in Technische Berichte des Instituts für Informatik, pages 41–50, Bad Honnef, August 2007. Christian–Albrechts–Universität, Kiel.

[Plü07b] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, Armin Wolf, and Joachim Baumeister, editors, Proceedings of 17th International Conference on Applications of Declartive Programming and Knowledge Management (INAP 2007) and 21th Workshop on (Constraint) Logic Programming (WLP 2007), number 434 in Technical Report, pages 234–245. Bayerische Julius–Maximillian–Universität Würzburg, Institut für Informatik, October 2007.

[Plü07c] 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.

[Plü12] Martin Plümicke. Complete type inference in Java 8, 2012. to appear.