DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart

Engineering

INSIGHTS

Schriftenreihe der Fakultät Technik: 1/2022

Java-TX: The language

Prof. Dr. habil. Martin Plümicke, Etienne Zink

Etienne Zink, Student im 6. Semester des Studiengangs Informatik, hat sich in seiner Studienarbeit mit dem Thema „Heterogene Übersetzung echter Funktionstypen in Java-TX“ beschäftigt. Die Studienarbeit fand im Rahmen der Forschung zur Programmiersprache Java-TX am Campus Horb statt.

Professor Martin Plümicke lehrt und forscht im Studiengang Informatik am Campus Horb der Dualen Hochschule Baden-Württemberg Stuttgart. Die Schwerpunkte liegen im Bereich Programmiersprachen. Gemeinsam mit seinen Doktoranden und Studierenden entwickelt er JAVA-TX als Erweiterung der Programmiersprache Java.

Duale Hochschule Baden-Württemberg Stuttgart – Campus Horb
Florianstraße 15
72160 Horb am Neckar

E-Mail: m.pluemicke@hb.dhbw-stuttgart.de

Zusammenfassung

Java-TX (TX steht für **T**ype e**X**tended) ist eine Erweiterung von Java, deren wesentliche neue Eigenschaften globale Typinferenz und echte Funktionstypen sind. Funktionstypen werden in Java-TX ähnlich wie bei Scala eingeführt. Zusätzlich werden sie in das Zieltypen-Konzept von Java integriert, wie es im sogenannten Stawman-Ansatz vorgeschlagen wurde. Diese Erweiterungen führen zu einer neuen Überladungseigenschaft, die Methodendeklaration mit Durchschnittstypen erlaubt. Darüber hinaus haben Java-TX-Methoden allgemeinste Typen. Zusätzlich werden Typparameter von Klassen und Methoden automatisch generiert. Schließlich werden Parameter von Funktionstypen heterogen übersetzt.

Abstract

Java-TX (TX standing for **T**ype e**X**tended) is an extension of Java. The predominant new features are global type inference and real function types for lambda expressions. These function types are introduced in a similar fashion as in Scala but additionally integrated into the Java target-typing as proposed in a so-called strawman approach. These extensions lead to a new, more powerful overloading mechanism, which means that the type of a method declaration could be an intersection of method types. Furthermore, there is a principal type property for Java-TX methods. Additionally, the type parameter of classes and their methods are generated automatically. Finally, the type parameters of function types are translated heterogeneously.

1 Introduction

Since version 1.5 the programming language Java has been extended by incorporating many features from functional programming languages. Version 1.5 saw the introduction of generics. Generics are known as parametric polymorphism in functional programming languages. In contrast to functional programming languages such as Haskell or SML, object-oriented languages like Java allow subtyping and states of objects. Therefore, the variance of type arguments has to be considered. In PIZZA, the first approach of parametric polymorphism in Java-like languages, the arguments were declared as invariant, which means for `Vector<T>` $\leq^*$[1] `Collection<T>` and `Integer` $\leq^*$ `Object` holds

`Vector<Integer>` $\leq^*$ `Collection<Integer>`

but neither covariance

`Vector<Integer>` $\leq^*$ `Collection<Object>`

nor contravariance

`Vector<Object>` $\leq^*$ `Collection<Integer>`

is correct. Invariance of type arguments is a strict restriction. Therefore, use-site variance by so-called wildcards was introduced in Java 5. In some cases, wildcards allow covariance or contravariance, respectively. In Java 8 lambda expressions were introduced, but not real function types. The types of lambda expressions are defined as target types, which are functional interfaces (essentially interfaces with one method).

Local type inference was introduced in the versions 5, 7, and 10. In Java 5 the automatic determination of parameter instance was introduced. In Java 7 the diamond operator was introduced. In

[1] $\leq^*$ stands for the subtyping relation.

```
source     := class*
class      := Class(cname, [generics,] extends( ctype ), fielddecl*, methoddecl²*)
ctype      := Type( cname, type* )
type       := ctype | tvar
generics   := (tvar, extends(tvar | Object))*
fielddecl  := Field( [type,]var[, expr] )
methoddecl := Method( [generics,][type,]mname, (var[: type])*, block )
block      := Block( stmt* )
stmt       := block | Return( expr ) | While( bexpr, stmt ) | LocalVarDecl( [type,]var )
            | If( bexpr, stmt[, stmt] ) | EmptyStmt | stmtexpr
lambdaexpr := Lambda( (var[: type])*, (stmt | expr) )
stmtexpr   := Assign( vexpr, expr ) | MethodCall( iexpr, mname, expr* ) | New( type )
vexpr      := LocalVar( var ) | InstVar( iexpr, var )
iexpr      := vexpr | lambdaexpr | stmtexpr | Cast( type, iexpr ) | this | super
expr       := iexp | bexp | sexp³
```

Figure 1: The abstract syntax of a core of Java-TX

Java 10, finally, the `var` keyword for types of local variables was introduced [7].

Up until now the features of global type inference (no type declarations are necessary without losing static typing property) and real function types have not been addressed in Java.

This paper is a summary of the previous research on Java-TX. After definition of the language (Sec. 2) we begin by expanding local type inference to global type inference (Sec. 3). We then extend the Java overloading mechanism of method names to overloaded method declarations for which we introduce intersection types (Sec. 4). We go on to give a principal type property for Java-TX methods and fields (Sec. 5). Additionally, we integrate target-typing for lambda expressions and real function type as described theoretically in the strawman approach [16, 30] (Sec. 6). We subsequently present the feature of generated generics of classes and its methods (Sec. 7). Finally, we integrate the heterogeneous translation of type parameters into Java-TX (Sec. 8).

2 The language

The language Java-TX is presented in Fig. 1. In this paper we treat Java-TX in an abstract representation of a core of Java 8. Beside some trivia we reduce the language by two essential features, exceptions and generics bound by non type-variable types (only type variables as bounds are allowed). Furthermore, basic types (`int`, `float`, `bool`, ...) were left out, such that the boxed variants have to be used. But the literals `1`, `2`, `3`, ..., `true`, `false` are still allowed. The optional type annotations [***type***] are the types, which can be inferred by our type inference algorithm (cp. Section 3).

The Java-TX type system correspond substantially to the original Java 8 type system as given in [10]. We extend the type system by introduction of real function types [29].

3 Global type inference for Java-TX

Global type inference allows us to leave out all type annotations. As in functional programming languages like Haskell, the compiler similarly determines a principal typing, such that Java-TX is statically typed as original Java. Let us first consider a simple example.

Example 3.1. *Let the class* `Fac` *(Fig. 2) be given. It is the simple iterative implementation of the factorial function. The return and the argument type of* `getFac` *are left out. The type inference algorithm has to infer the types. The types are determined by the declaration of* `res` *and the overloaded operator * . In order to reduce the complexity of the type inference algorithm for overloaded operators in the same way as for overloaded methods, only types are inferred which are explicitly imported by the keyword* **import**. *Therefore for* `getFac` *the typing*

```
import java.lang.Integer;

class Fac {

    getFac(n){
        var res = 1;
        var i = 1;
        while(i<=n) {
            res = res * i;
            i++;
        }
        return res;
    }
}
```

Figure 2: The class `Fac`

```
java.lang.Integer
    getFac(java.lang.Integer n)
```

is inferred.

Before we consider a more complex example we shall offer a formal definition of typed identifiers (variables and method names) in Java-TX.

Definition 3.2 (Types of identifiers)**.** *For* identifiers *in Java-TX programs the* types *are defined as follows:*

$v : \theta$**:** *Type of local variable* v*.*

$cl\langle\overline{T < T'}\rangle.f : \theta$**:** *Type of a field* f *of the class* cl *with the generics* $\overline{T}$ *bound by* $\overline{T'}$*.*

$cl\langle\overline{T < T'}\rangle.m : \langle\overline{R < R'}\rangle(\theta_1, \ldots, \theta_n) \to \theta$**:** *Type of a method* m *with generics* $\overline{R}$ *bound by* $\overline{R'}$ *of the class* cl *with the generics* $\overline{T}$ *bound by* $\overline{T'}$*.*

$$cl\langle\overline{T < T'}\rangle.m : \langle\overline{R_1 < R'_1}\rangle((\theta_{1,1}, \ldots, \theta_{1,n}) \to \theta_1) \;\&\ldots\&\; \langle\overline{R_m < R'_m}\rangle((\theta_{m,1}, \ldots, \theta_{m,n}) \to \theta_m):$$

Intersection type of the overloaded method m*.*

Let us consider a second more complex example.

Example 3.3. *The program in Fig. 3 is given. The class* `Matrix` *is implemented as an extension of* `Vector<Vector<Integer>>`*. The method* `mul` *implements the multiplication of two matrices. An obvious typing of* `mul` *would be*

```
Matrix mul(Matrix m)
```

[4] With $\overline{x}$ we denote in this paper tuples $(x_1, \ldots, x_n)$ of types, variables, terms, etc.

```
import java.util.Vector;

class Matrix
        extends Vector<Vector<Integer>> {
  mul(m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg
                  + v1.elementAt(k)
                  * m.elementAt(k)
                      .elementAt(j);
              k++; }
            v2.addElement(erg);
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

Figure 3: The class `Matrix`

The question is whether this typing is the only possible typing. If not, then the question to be asked is whether it is the principal *typing. It is easy to see that there are other correct typings, e.g.*

```
Matrix mul(Vector<Vector<Integer>> m)
```

or

```
Vector<Vector<Integer>>
        mul(Vector<Vector<Integer>> m)
```

are correct. But

```
Matrix mul(Vector<? extends
          Vector<? extends Integer>> m)
```

is also correct. We collect all correct typings of `mul` *to an intersection type of* `mul`*, given in Fig. 4.*

We remember the usual subtyping definition for function types.

Definition 3.4 (Subtyping relation $\leq^*$ on function types)**.** *For two given functions types*

$$(\tau_1, \ldots, \tau_n) \to \tau_0 \quad \textit{and} \quad (\theta_1, \ldots, \theta_n) \to \theta_0$$

the following is valid:

$$(\tau_1, \ldots, \tau_n) \to \tau_0 \leq^* (\theta_1, \ldots, \theta_n) \to \theta_0$$

iff $\theta_i \leq^* \tau_i$ *and* $\tau_0 \leq^* \theta_0$*.*

```
Matrix.mul : Matrix → Matrix
           & Matrix → Vector<Vector<Integer>>
           & Vector<Vector<Integer>> → Matrix
           & Vector<Vector<Integer>> → Vector<Vector<Integer>>
           & Vector<Vector<? extends Integer>> → Matrix
           & Vector<Vector<? extends Integer>> → Vector<Vector<Integer>>
           & Vector<? extends Vector<Integer>> → Matrix
           & Vector<? extends Vector<Integer>> → Vector<Vector<Integer>>
           & Vector<? extends Vector<? extends Integer>> → Matrix
           & Vector<? extends Vector<? extends Integer>> → Vector<Vector<Integer>>
           & Matrix → Vector<? extends Vector<? extends Integer>>
           & ...
```

Figure 4: Intersection type of `mul`

Regarding this definition, a first line of thought in defining a *principal* type is the intersection type of all minimal elements of these function types.

$$\texttt{Vector<}_{?}\texttt{Vector<}_{?}\texttt{Integer>>}\rightarrow\texttt{Matrix}$$ [5]

is the subtype of all other types (minimum in the subtyping relation). This means that the domain of the function is maximal and the range minimal. Therefore this should be the principal type of `mul`.

Before we go on to offer a formal definition of the principal type in Section 5, we shall consider the type inference algorithm and the extended overloading mechanism of Java-TX.

3.1 The type inference algorithm

Here we would like to present a short overview of the type inference algorithm. The input is the abstract syntax tree of the corresponding Java class. The type inference algorithm consists of two steps:

Tree traversing: In a traversing of the abstract syntax tree, a type is mapped to each node of the methods' statements and expressions. If the corresponding types are left out, a type variable is mapped as type placeholder. Otherwise, the known type is mapped.

During the traversing a set of type constraints $\{\overline{ty \lessdot ty'}\}$ is generated. The constraints represent the type conditions as defined in the Java specification [10]. For more details see the function **TYPE** in [27].

[5] ${}_{?}Type$ and ${}^{?}Type$ are abbreviations for `? extends` $Type$ and `? super` $Type$, respectively.

Type unification: For the set of type constraints $\{\overline{ty \lessdot ty'}\}$ general unifiers (substitution) σ are demanded, such that

$$\overline{\sigma(ty1) \leq^* \sigma(ty')}.$$

The result of the type unification is a set of pairs

$$(\{\overline{(T \lessdot T')}\}, \sigma),$$

where $\{\overline{(T \lessdot T')}\}$ is a set of remaining constraints consisting of two type variables and σ is a general unifier.

The type unification algorithm is given in [26, 33]. There we proved that the unification is indeed not unitary, but finitary, meaning that there are finitely many general unifiers.

Let us consider the application of the type inference algorithm to the factorial example (Example 3.1). Note that we omit the `import` statements for the sake of readability in the further examples.

Example 3.5. *First, we present the essential type variables which are mapped to nodes of the method* `getFac`*:*

```
class Fac {
  N getFac(O n) {
    P res = 1;
    R i = 1;
     while((i::R) <= (n::O))::T  {
        (res::P)=((res::P)*(i::R))::U;
        (i::R)++;
      }
      return(res::P);
  }
}
```

The generated constraints are

$\{(P \doteq N), (U \lessdot P), (O \lessdot \texttt{java.lang.Number}),$
$(R \lessdot \texttt{java.lang.Number},$

$(\texttt{java.lang.Boolean} \doteq T),$
$(\texttt{java.lang.Integer} \doteq U),$
$(R \lessdot \texttt{java.lang.Integer}),$
$(P \lessdot \texttt{java.lang.Integer})\,\}$

The result of type unification is given as:

$$\{\,(\emptyset, [(U \mapsto \texttt{java.lang.Integer}),\\ (P \mapsto \texttt{java.lang.Integer}),\\ (R \mapsto \texttt{java.lang.Integer}),\\ (O \mapsto \texttt{java.lang.Integer}),\\ (N \mapsto \texttt{java.lang.Integer}),\\ (T \mapsto \texttt{java.lang.Boolean})]\,\}$$

In this example, no constraints consisting only of type variables remain. Furthermore, there is only one general unifier.

If we instantiate the type variables by the determined types, we get:

```
class Fac {
  Integer getFac(Integer n) {
    Integer res = 1;
    Integer i = 1;
    while((i::Integer)
                <= (n::Integer))::Boolean
    {
      (res::Integer) =
        ((res::Integer) * (i::Integer))
        ::Integer;
      (i::Integer)++;
    }
    return(res::Integer); } }
```

The unification in the above example has one solution. If there would be more than one solution, there could be more than one principal typing of the method. We consider this in Example 4.1.

The set of remaining constraints which consist only of type variables is empty. If this set would not be empty, then type parameters (generics) of the class or of its method would be generated. We consider this in Section 7.

4 Overloading

The following example serves to show the overloading mechanism.

Example 4.1. *Let the classes* `OL` *and* `OLMain` *be given.*

```
class OL {
  m(Integer x) { return x + x; }

  m(Boolean x) { return x || x; }
}


class OLMain {
  main(x) {
    var ol = new OL();
    return ol.m(x); } }
```

This example illustrates the extended overloading mechanism of Java-TX. In the class `OL` *the method name* `m` *is overloaded by two different method declarations as in standard Java. In the class* `OLMain` *an instance of* `OL` *is created and on the instance the overloaded method* `m` *is called. This means the only method declaration* `main` *is overloaded by both types. The type of* `main` *is then given as the intersection type:*

$$\begin{array}{rcl} \texttt{OLMain.main} & : & \texttt{Integer} \to \texttt{Integer} \\ & \& & \texttt{Boolean} \to \texttt{Boolean} \end{array}$$

If we leave out the type annotations in `OL` *and add some import declarations:*

```
import java.lang.Integer;
import java.lang.Double;
import java.lang.String;
import java.lang.Boolean;

class OL {
  m(x) { return x + x; }

  m(x) { return x || x; }
}
```

the type of the first method `m` *is:*

$$\begin{array}{rcl} \texttt{OL.m} & : & \texttt{Integer} \to \texttt{Integer} \\ & \& & \texttt{Double} \to \texttt{Double} \\ & \& & \texttt{String} \to \texttt{String}, \end{array}$$

as `+` *is an overloaded operation symbol. The second method* `m` *has the unchanged type*

$$\texttt{OL.m} \;:\; \texttt{Boolean} \to \texttt{Boolean}.$$

This means that the `main` *declaration is overloaded by all four types of both methods* `m`*:*

$$\begin{array}{rcl} \texttt{OLMain.main} & : & \texttt{Integer} \to \texttt{Integer} \\ & \& & \texttt{Double} \to \texttt{Double} \\ & \& & \texttt{String} \to \texttt{String} \\ & \& & \texttt{Boolean} \to \texttt{Boolean} \end{array}$$

This example shows the extended overloading mechanism in Java-TX. Standard-Java only allows the overloading of method identifiers, meaning that multiple method declarations with the same identifier have to be declared (cp. method `m` with explicitly declared types in the class `OL`). In contrast, in Java-TX one declaration could be overloaded, which means that one declaration has different types (cp. method `main` in the class `OLMain`).

In the following section we shall define a principal type for Java-TX programs which is the result of the type inference algorithm.

5 Principal type

In [6] for functional programs (without subtyping) a *principal type* is defined as:

A type scheme for a declaration is a principal type scheme*, if any other type scheme for the declaration is a generic instance of it.*

We combine this definition of principal type scheme for functional programs with the idea that the principal types are the minimal elements in the subtyping relation:

An intersection type scheme with a minimal number of elements for a declaration is a principal type scheme*, if any other type scheme for the declaration is a supertype of a generic instance of one element of the intersection type scheme.*

Following this definition, in Example 3.3

```
Matrix.mul :
      Vector<?Vector<?Integer>>→Matrix
```

and in Example 4.1

```
OLMain.main  :  Integer → Integer
             &  Double → Double
             &  String → String
             &  Boolean → Boolean
```

are principal types.

The following example shows that this definition of the Java-TX principal type has to be refined if we consider overlapping arities of methods.

Example 5.1. *Given the following Java-TX program:*

```
import java.util.Vector;
import java.util.Stack;

class Put {
  <T> putElement(T ele, Vector<T> v) {
    v.addElement(ele);
  }

  <T> putElement(T ele, Stack<T> s) {
    s.push(ele);
  }

  main(ele, x) {
    putElement(ele, x);
  }
}
```

The inferred intersection type of `main` *is*

$$\begin{aligned}\texttt{Put.main} : (\texttt{T}, \texttt{Vector<T>})&\rightarrow\texttt{void}\\ \&\,(\texttt{T}, \texttt{Stack<T>})&\rightarrow\texttt{void}.\end{aligned}$$

With the given definition the principal type would be $(\texttt{T}, \texttt{Vector<T>})\rightarrow\texttt{void}$*. This is not correct as the stack's application would disappear. Therefore, we have to refine the definition by considering its call graphs.*

Definition 5.2 (Principal type of Java methods)**.** *An intersection type of a method* `m` *in a class* `cl`

$$\begin{aligned}\texttt{cl}\texttt{<}\overline{T < T'}\texttt{>}.\texttt{m} : \texttt{<}\overline{R_1 < R_1'}\texttt{>}&((\theta_{1,1},\ldots,\theta_{1,n})\rightarrow\theta_1)\\ \&\ldots\&&\\ \texttt{<}\overline{R_m < R_m'}\texttt{>}&((\theta_{m,1},\ldots,\theta_{m,n},)\rightarrow\theta_m)\end{aligned}$$

is called principal *if the number of elements of the intersection is minimal and for any correct type annotated method declaration*

$$rty\ \texttt{m}(ty1\ \texttt{a1},\ldots,tyn\ \texttt{an})\ \{\ldots\}$$

there is an element $((\theta_{i,1},\ldots,\theta_{i,n},)\rightarrow\theta_i)$ *of the intersection type and there is a substitution* σ*, such that*

$$\sigma(\theta_i) \leq^* rty, ty1 \leq^* \sigma(\theta_{i,1}), \ldots, tyn \leq^* \sigma(\theta_{i,n})$$

and the call graphs of $(\theta_{i,1},\ldots,\theta_{i,n},)\rightarrow\theta_i$ *and* $(ty1,\ldots,tyn)\rightarrow rty$ *are equal.*

Example 5.3. *Continuing Example 5.1 the principal type of* `main` *is*

$$\begin{aligned}\texttt{Put.main} : (\texttt{T}, \texttt{Vector<T>})&\rightarrow\texttt{void}\\ \&\,(\texttt{T}, \texttt{Stack<T>})&\rightarrow\texttt{void}\end{aligned}$$

as the call graphs of $(\texttt{T}, \texttt{Vector<T>}) \rightarrow \texttt{void}$ *and* $(\texttt{T}, \texttt{Stack<T>})\rightarrow\texttt{void}$ *differ (cp. Fig. 5).*

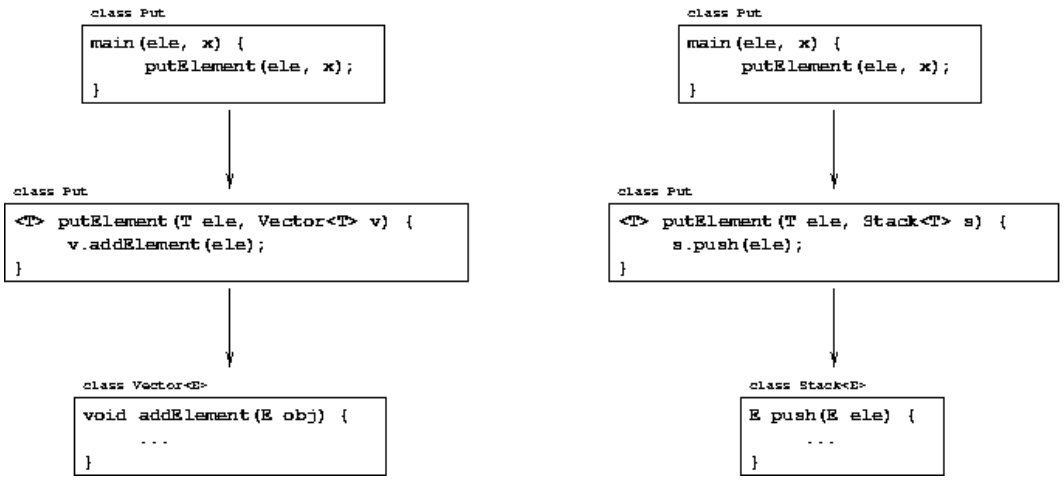

Figure 5: Call graphs of the method `main` in the class `Put`

A detailed overview of call graphs and the corresponding algorithm can be found in [25].

Finally, we give the definition of the principal type of fields in Java-TX classes. It is possible to assign a lambda expression to a field of a Java class. Therefore it make sense to define a principal type for fields.

Definition 5.4 (Principal type of Java fields)**.** *A type θ of a field* `f` *in a class* `cl`

$$cl\overline{<T < T'>}.\mathtt{f} : \theta$$

is called principal *if for any type correct field declaration*

```
ty f;
```

there is a substitution σ, such that

$$\sigma(\theta) \leq^* \mathtt{ty}$$

is valid.

As fields cannot be overloaded no intersection types are necessary.

5.1 Type inference and principal type

Principal types in Java-TX can be inferred for nearly all methods. There are only two restrictions. On the one hand type inference is undecidable in the presence of polymorphic recursion as in all languages with global type inference following the approach of [6]. On the other hand, Java type-check become undecidable in the presence of F-bounded polymorphism in combination with covariance and contravariance by bound wildcards. This is shown in [11]. There is even a reduction from the halting problem of Turing machines to subtype checking in Java.

The next example shows the consequence of polymorphic recursion.

Example 5.5. *Let the class* `Map` *be given:*

```
class Map {

  <T> List<T>
    map (Function<T,T> f, List<T> l){
      for(int i=0; i < l.size(); i++) {
        l.set(i, f.apply(l.get(i)));
        return l;
    }

    List<Integer>
      addInt5ToList (List<Integer> l) {
        return map(x -> x+5, l);
    }

    List<String>
      addStr5ToList (List<String> l) {
        return map(x -> x + "5", l);
    }
}
```

If we leave out all type annotations of the methods in the class `Map` *the type inference goes wrong due to the fact that the type variable* `T` *could either be instantiated by* `Integer` *or by* `String` *but not by both at once.*

The difference between the explicitly typed class and the typeless class in this example corre-

sponds to the difference of the Damas-Milner type system [6] and the Mycroft-Milner type system [19], where Henglein reduces the type inference problem of the Mycroft-Milner type system to semi-unification [12] and semi-unification has been shown to be recursively undecidable [15].

F-bounded polymorphism in combination with bound wildcards in Java allows recursion in the type arguments of parametrized type. E.g. for

```
class Integer implements
            Comparable<Integer> {...}
```

it holds true that:

$$\begin{aligned}&\texttt{Integer} \leq^* \texttt{Comparable<Integer>}\\ &\leq^* \texttt{Comparable<}_?\texttt{Integer>}\\ &\leq^* \texttt{Comparable<}_?\texttt{Comparable<Integer>>}\\ &\leq^* \texttt{Comparable<}_?\texttt{Comparable<}_?\texttt{Integer>>}\\ &\leq^* \texttt{Comparable<}_?\texttt{Comparable<}_?\texttt{...>>}\\ &\leq^* \ldots\end{aligned}$$

and

$$\begin{aligned}&\ldots\\ &\leq^* \texttt{Comparable<}^?\texttt{Comparable<}^?\texttt{...>>}\\ &\leq^* \texttt{Comparable<}^?\texttt{Comparable<}^?\texttt{Integer>>}\\ &\leq^* \texttt{Comparable<}^?\texttt{Comparable<Integer>>}\\ &\leq^* \texttt{Comparable<}^?\texttt{Integer>}\end{aligned}$$

This means that for instance the constraint $\texttt{Integer} \lessdot T$ as well as the constraint $T \lessdot \texttt{Comparable<? super Integer>}$, respectively, have infinite result sets, respectively.

In Java-TX the problem is solved by capping the infinite chains. This means if a type is contained in an argument of a supertype or of a subtype, respectively, then these types and all supertypes or all subtypes, respectively, are excluded.

In the first constraint $\texttt{Integer} \lessdot T$ the type `Integer` is contained in its supertype `Comparable<Integer>`. Therefore `Comparable<Integer>` and all supertypes of `Comparable<Integer>` are excluded. In the second constraint $T \lessdot \texttt{Comparable<}^?\texttt{Integer>}$ the type $\texttt{Comparable<}^?\texttt{Integer>}$ is contained in $\texttt{Comparable<}^?\texttt{Comparable<}^?\texttt{Integer>>}$. Therefore the type $\texttt{Comparable<}^?\texttt{Comparable<}^?\texttt{Integer>>}$ and its subtypes are excluded.

Following this, the type inference in the class `IntComparable`

```
class IntComparable {
    void m(k) {
        k = 1;
    }
}
```

determines the type `Integer` for the argument `k` as all other correct types `Comparable<Integer>`, `Comparable<? extends Integer>`, ... are excluded.

6 Real function types

In Java 8, lambda expression has indeed been introduced, but not function types. Instead, there are functional interfaces as target types of lambda expressions. There are many disadvantages because of the lack of function types. Java-TX counters these disadvantages by introducing function types in a similar way as in Scala without losing the convenience of functional interfaces as target type of lambda expressions [29]. Java-TX implements the so-called strawman approach, which was theoretically given in [16, 30].

In Java 8 function types are simulated in the package `java.util.function`:

```
public interface Function<T,R> {
  R apply(T t);
}
```

```
public interface BiFunction<T,U,R> {
  R apply(T t, U u);
}
```

There are some inconveniences.

Subtyping

Although the following holds true for subtypes (cp. Def. 3.4)

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

for the functional interface `Function`

$$\texttt{Function<}T'_1,T_0\texttt{>} \leq^* \texttt{Function<}T_1,T'_0\texttt{>}$$

for $T_i \lneq^* T'_i$, is not correct, as Java has use-site variance. Therefore, arguments of types without wildcards are invariant.

Example 6.1. *For* $\texttt{Integer} \leq^* \texttt{Number} \leq^* \texttt{Object}$ *it holds true that:*

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

but

```
Function<Number, Number> f_NN = ...
Function<Integer, Object> f_IO = f_NN
```

*is **wrong**, as*

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

This problem could be solved by wildcards. It holds true for $T_i \leq^* T_i'$:

$$\texttt{Function<}T_1', T_0\texttt{>} \leq^* \texttt{Function<? super } \texttt{T}_1, \texttt{? extends } \texttt{T}_0'\texttt{>},$$

for $T_i \leq^* T_i'$. This means

```
Function<? Integer, ? Object> f_IO= f_NN
```

is correct.

Direct application of lambda expressions

In the λ-calculus β-conversion (direct application of a lambda expression to its arguments), the following is possible:

$$(\lambda\, x.E)arg = E[x/arg].$$

In Java 8 this lambda term would have the following form:

```
(x -> h(x)).apply(arg);
```

Such expressions are not permitted. As the lambda expression has no explicit type, it is not obvious if the method apply exists at all. This problem could be solved by introducing a type cast:

```
((Function<T,R>)x -> h(x)).apply(arg);
```

Summary

The drawbacks of the lack of function types are all solved in Java 8:

Missing function types: The function types are replaced by the functional interfaces `Bi/Function` in the package `java.util.function`.

Subtyping problem: The problem of the functional interface's behaviour differing from the usual definition of subtyping is solved by using wildcards.

Impossibility of direct application of lambda expressions: The impossibility of applying a lambda expression directly to its arguments is solved by using type casts.

This means all problems are solvable, but the solutions are not attractive. We therefore introduced real function types in Java-TX. We extended Java by two sets of special functional interfaces with declaration-site variance type parameters:

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

interface FunVoidN$$<-T1, ..., -TN> {
  void apply(T1 arg1, ... , TN argN);
}
```

where

- Fun*N$$*<T_1'...T_N',T_0> $\leq^*$ Fun$N$$$<T_1...T_N,T_0'> iff $T_i \leq^* T_i'$
- In Fun*N$$* no wildcards are allowed.

The Lambda-expressions are explicitly typed by Fun*N*$$-types.

Example 6.2. *Let us consider the changed matrix program in Fig. 7.*

In Java 8 there are two possibilities to type the field `mul`:

- Using `java.util.function.*`:

  ```
  mul:BiFunction<? Vector<? Vector<? Integer>>
                 ? Vector<? Vector<? Integer>>
                 ? Matrix>
  ```

  This type declaration is less readable. In particular, the mixture of super and extends wildcards in the second argument are extremely unusual.

```
interface MatrixOperation {
    MatrixOP apply(Vector<? extends Vector<? extends Integer>> arg1,
                   Vector<? extends Vector<? extends Integer>> arg2);
```

Figure 6: Interface `MatrixOperation`

```
class MatrixOP
      extends Vector<Vector<Integer>> {
  mul = (m1, m2) -> {
    var ret = new MatrixOP();
    var i = 0;
    while(i < m1.size()) {
      var v1 = m1.elementAt(i);
      var v2 = new Vector<Integer>();
      var j = 0;
      while(j < v1.size()) {
        var erg = 0;
        var k = 0;
        while(k < v1.size()) {
          erg = erg
              + v1.elementAt(k)
              * m2.elementAt(k)
                  .elementAt(j);
          k++; }
        v2.addElement(erg);
        j++; }
      ret.addElement(v2);
      i++;}
    return ret;}
```

Figure 7: Matrix with lambda expression

- Defining an own functional interface MatrixOperation (Fig. 6):

  ```
  mul:  MatrixOperation
  ```

  This type declaration is very short, but the type `MatrixOperation` hides the most information.

In contrast, Java-TX infers the function type:

```
mul:Fun2$$<Vector<?Vector<?Integer>>
           Vector<?Vector<?Integer>>
           Matrix>
```

This type is indeed complex, too. But the arguments of the function type `Fun2$$` have no wildcards. This reduces the confusion.

7 Generalized type variables

In a similar way as in type inference of functional programming languages, free type variables which are not instanced by other types after type inference are generalized to generics. In comparison to functional programming languages, in Java subtyping leads to a more powerful generalization mechanism.

Keeping in mind the result of the type unification (Sec. 3.1) given as a set of pairs of

- remaining constraints consisting only of pairs of type variables and a
- most general unifier:

$$\{(\{\overline{T_1 \lessdot T_1'}\}, \sigma_1),\ \ldots,\ (\{\overline{T_n \lessdot T_n'}\}, \sigma_n)\}.$$

In the previous sections we considered the unifiers. In this section we shall consider the remaining constraints. In the existing type inference algorithm of functional programming languages without subtyping (e.g. Haskell or SML) the remaining type variables are generalized such that any type can be instantiated if the function is used.

Following up this idea, the remaining type variables become bound type parameters of the class and its methods, respectively, where the left-hand side of a constraint is a type parameter and the right-hand side is its bound.

There are three possibilities for adapting this concept to Java:

- All in the constraints, connected type variables are mapped to one type parameter.
- All constraints are transferred to bound type parameters of the class.
- The constraints are transferred to bound type parameters of the class and its methods.

We shall see that the third possibility leads to the principal types. Additionally, due to the Java restrictions of type parameters, some type parameters have to be collected to one new type parameter. As proposed in the first possibility.

This section is structured as follows: After a motivating example, we present an apportioning of the type variables to the class and its methods. We then reduce the respective set of type variables

```
class TPHsAndGenerics {
    id = x -> x;

    id2 (x) {
        return id.apply(x);}

    m(a, b) {
        var c = m2(a,b);
        return a; }

    m2(a, b){
        return b; } }
```

```
class TPHsAndGenerics {
    Fun1$$<UD, ETX> id = (DZP x) -> x;

    ETX id2(V x) {
        return id.apply(x);}

    AB m(AB a, AD b){
        AE c = m2(a,b);
        return a; }

    AI m2(AM a, AI b){
        return b; } }
```

Figure 8: Class `TPHsAndGenerics` before and after type inference

such that the Java restrictions of type variables are fulfilled.

Let us start with a motivating example.

Example 7.1. *On the left in Fig. 8 a Java-TX program is given. The identity function is mapped to the field* `id`*. In the method* `id2` *the identity function is called. In the method* `m` *the method* `m2` *is called.*

The application of the type inference algorithm is presented on the right, and the remaining set of constraints of the type unification is:

$$cs = \{\texttt{AD} \lessdot \texttt{AI},\ \texttt{V} \lessdot \texttt{UD},\ \texttt{AI} \lessdot \texttt{AE},\ \texttt{AB} \lessdot \texttt{AM},\ \texttt{DZP} \lessdot \texttt{ETX},\ \texttt{UD} \lessdot \texttt{DZP}\}.$$

7.1 Family of generated generics

We divide up the set of remaining constraints cs by transferring it to a family cs' where the index set is given as the class name and its method names.

Definition 7.2 (Family of generated generics)**.** *The* family of generated generics *is defined as*

$$cs' = (cs'_{in})_{in \in CLM},$$

where

$$CLM = \{\, cl \,\} \cup \{\, m \mid m \ is\ method\ in\ cl \,\}$$

is the index set of the class name and its methods' names.

Let cs be a set of remaining constraints as result of the type unification. cs is transferred to the family of generated generics cs' where, the set of generated generics of the class cs'_{cl} are given as:

- *all type variables of the fields with its bounds*
- *the closure of all bounds of type variables of the fields with its bounds, and*
- *all unbound type variables of the fields and all unbound bounds with* `Object` *as bound.*

The set generated generics cs'_m of its methods m: are given, respectively, as:

- *the type variables of the method m with its bounds, where the bounds are also type variables of the method,*
- *new constructed pairs $T_1 \lessdot T_2$ of type variable T_1, T_2 of the method m which are in the transitive closure*

  $$T_1 \lessdot R_1 \leq^* R_2 \lessdot T_2$$ [6]

  where $R_1 \leq^* R_2 \in cl'_{m'}$ and m' is called in m,
- *all type variables of the method m with its bounds, where the bounds are type variables of fields and*
- *all unbound type variables of the method m and all unbound bounds with* `Object` *as bound.*

We present the family of generated generics for the class `TPHsAndGenerics` from Example 7.1

Example 7.3. *The set of remaining constraints*

$$cs = \{\texttt{AD} \lessdot \texttt{AI}, \texttt{V} \lessdot \texttt{UD}, \texttt{AI} \lessdot \texttt{AE}, \texttt{AB} \lessdot \texttt{AM}, \texttt{DZP} \lessdot \texttt{ETX}, \texttt{UD} \lessdot \texttt{DZP}\}$$

of the class `TPHsAndGenerics` *results in the family of generated generics*

[6] $\leq^*$ stands for the reflexive and transitive closure of $\lessdot$.

The set of generated generics $cs'_{\texttt{TPHsAndGenerics}}$ of the class:

- *Type variables of the fields with its bounds:*

$$\{\, \texttt{UD} \lessdot \texttt{DZP} \,\}$$

- *Closure of all bounds of type variables of the fields with its bounds:*

$$\{\, \texttt{DZP} \lessdot \texttt{ETX} \,\}$$

- *All unbound type variables of the fields and all unbound bounds with* `Object` *as bound:*

$$\{\, \texttt{ETX} \lessdot \texttt{Object} \,\}$$

The set of generated generics $cs'_{\texttt{id2}}$:

- *All pairs where the bounds are type variables of fields:*

$$\{\, \texttt{V} \lessdot \texttt{UD} \,\}$$

The set of generated generics $cs'_{\texttt{m}}$:

- *New pair*

$$\{\, \texttt{AD} \lessdot \texttt{AE} \,\}$$

  which is in the transitive closure $\texttt{AD} \lessdot \texttt{AI} \lessdot \texttt{AE}$, *where* $\texttt{AI} \in cl'_{\texttt{m2}}$ *and* `m2` *is called in* `m`.

- *All unbound type variables of the method* `m` *and all unbound bounds with* `Object` *as bound:*

$$\{\, \texttt{AB} \lessdot \texttt{Object}, \texttt{AE} \lessdot \texttt{Object} \,\}$$

The set of generated generics $cs'_{\texttt{m2}}$:

- *All unbound type variables of the method m with* `Object` *as bound:*

$$\{\, \texttt{AM} \lessdot \texttt{Object}, \texttt{AI} \lessdot \texttt{Object} \,\}$$

The mapping of the family to the class and its methods in the Java-TX program is presented in Fig. 9, where the bounds `Object` *are left out.*

```
class TPHsAndGenerics
        <UD extends DZP, DZP extends ETX, ETX> {

    Fun1$$<UD, ETX> id = x -> x;

    <V extends UD> ETX id2(V x) {
        return id.apply(x);}

    <AD extends AE, AB, AE>
    AB m(AB a, AD b){
        AE c = m2(a,b);
        return a;}

    <AM, AI> AI m2(AM a, AI b){
        return b;}
}
```

Figure 9: Generated generics of the class `TPHsAndGenerics`

7.2 Java-conforming binary relation of type parameters

The set of remaining constraints as well as each element of the family of generated generics are arbitrary binary relations.

There are two conditions in Java which all members of the family of generated generics have to fulfill:

- The reflexive and transitive closure must be a partial ordering (the subtyping relation is a partial ordering).
- Two different elements have no common infimum (multiple inheritance is prohibited).

Consider the following lemma:

Lemma 7.4. *The reflexive and transitive closure of any binary relation is a partial ordering if it contains no cycle.*

Proof. A partial ordering is binary relation with the properties *reflexivity*, *transitivity*, and *antisymmetry*. The first two properties are given as we consider a reflexive and transitive closure. Therefore a reflexive and transitive closure is no partial ordering only if the property antisymmetry is not given. Antisymmetry is given if and only if there are no cycles. □

This means that we have to eliminate cycles and infima to get Java-conforming binary relations. We will do this by a surjective mapping of connected type variables to a new type variable.

First, we shall consider two examples which result in non-conforming relations.

Example 7.5 (Cycle)**.** *Let the class* `Cycle` *be given:*

```
class Cycle {
    m(x, y) {
        y = x;
        x = y;
    }
}
```

For the inferred method parameter `m(L x, M y)` *we get*

$$cs'_{\texttt{m}} = \{\, (L \lessdot M), (M \lessdot L) \,\}$$

But

```
<L extends M,M extends L>
                  void m(L x, M y) {...}
```

is not a correct declaration.

Example 7.6 (Infimum)**.** *Let the class* `Infimum` *be given:*

```
class Infimum {
    m(x, y, z) {
        y = x;
        z = x;
    }
}
```

For the inferred method parameter `m(L x, M y, N x)` *we get*

$$cs'_{\texttt{m}} = \{(L \lessdot M), (L \lessdot N), (M \lessdot \texttt{Object}), (N \lessdot \texttt{Object})\,\}$$

But

```
<L extends M, L extends N, M, N>
          void  m(L x, M y, N z) {...}
```

is not a correct declaration.

The general approach for eliminating cycles and infima is to equalize elements by a surjective map h that preserves the subtype relation:

For $T \leq^* T'$

$$h(\,T\,) \leq^* h(\,T'\,)$$

holds true.

We shall now present an algorithm which eliminate cycles and infima.

Algorithm 7.7 (Java-conforming relation)**.**

Input: *A member of the family of generated generics C.*

Output: *An adapted member of the family of generated generics C and a surjective mapping h that describes the adaption of C.*

Postcondition: *C is a minimal adaption such that it is Java-conforming.*

The algorithm:

1. ***Remove cycles:***

   For any $(T \lessdot K \lessdot G \lessdot ... \lessdot T)$ in C:

   - *Substitute all type variables of the cycle with a new type variable X in C.*
   - *Remove all constraints that built the cycle from C.*
   - *In h all removed type variables of the cycle are mapped to X.*

2. ***Eliminate infima:*** Apply the following steps until no infima are in C*:*

   For any

   $$\mathit{Constr}_T = \{\, (T \lessdot R), (T \lessdot S), \dots \,\} \subseteq C$$

   - *Create a new type variable X and create a new constraint $(T \lessdot X)$.*
   - *Add the new constraint $(T \lessdot X)$ to C.*
   - *Remove all constraints Constr_T from C.*
   - *All right-hand sides R of all constraints of Constr_T are mapped to X in h and substituted in C with X.*

Lemma 7.8. *Let cs'_m be a member of the family of generated generics and h the corresponding surjective map defined by Algorithm 7.7. For $T \leq^* T'$ holds true $h(\,T\,) \leq^* h(\,T'\,)$.*

Proof. For the removed cycles

$$T \lessdot K \lessdot G \lessdot ... \lessdot T$$

holds true

$$h(\,T\,) = h(\,K\,) = h(\,G\,) = ... = h(\,T\,).$$

For the eliminated infima $(T \lessdot R), (T \lessdot S)$ holds true $h(\,S\,) = h(\,R\,)$ and $(h(\,T\,) \lessdot h(\,R\,)), (h(\,T\,) \lessdot h(\,S\,))$. □

In the following examples, we apply the algorithm to the classes `Cycle` (Example 7.5) and `Infimum` (Example 7.6).

Example 7.9. *Applying the algorithm to the class* `Cycle` *we get the surjective mapping h with*

$$h(\,L\,) = X$$
$$h(\,M\,) = X$$

and the adapted class:

```
class Cycle {
    <X> void  m(X x, X y) {
        y = x;
        x = y;
    }
}
```

Example 7.10. *Applying the algorithm to the class* `Infimum` *we get the surjective mapping* h *with*

$h(\,M\,) = X$
$h(\,N\,) = X$

```
class Infimum {
  <L extends X, X>
    void  m(L x, X y, X z) {
        y = x;
        z = x;
    }
}
```

7.3 Further simplifications

Let us consider again the generated generics in the class `TPHsAndGenerics` (Figure 9). There is a class type parameter `DZP` that is never used. Additionally, in the field `id`, the argument and the result type differ although the identity functions return the argument `x`. It is not clear whether this is necessary.

Eliminate inner type variables

In the class `TPHsAndGenerics` the type variable `DZP` occurs in the type `Fun1$$<DZP,DZP>` of the lambda expression `x -> x` as inner type variable which is generated during the tree traversing of the type inference (cp. Sec. 3.1). As type variables of inner nodes are not needed to be declared, we eliminate type variables of inner nodes. In this step, the properties of the partial ordering have to be preserved.

Therefore, in the above example `UD` $\lessdot$ `ETX` follows from `UD` $\lessdot$ `DZP` $\lessdot$ `ETX`.

Equalize type variables in contravariant and covariant position

For the consideration of type variables like `UD` and `ETX`, we need another definition.

Definition 7.11 (Type variables in covariant and contravariant position)**.** *A type variable of an argument of a function/method is in* contravariant position. *A type variable of a return type of a function/method is in* covariant position.

In the type `Fun2$$<UD,ETX>` of the field `id` in the class `TPHsAndGenerics` the type variable `UD` is in contravariant position and `ETX` in covariant position.

Lemma 7.12. *Let* T *be a type variable in contravariant position and* U *a type variable in covariant position. If* $T \lessdot U$ *is valid, then in the sense of principality of the program the two type variables* T *and* U *can be equalized.*

Proof. From the definition of a principal type (Def. 5.2) it follows that the argument types have to be maximal and the return type has to be minimal. This means that T has to be maximal and U has to be minimal. Therefore, it follows from $T \lessdot U$ that for the principal type, $T = U$. □

In the above example, the type variables `UD` and `ETX` can be equalized. Fig. 10, below, presents the complete simplified class `TPHsAndGenerics`. Here, the inner type variables are eliminated and type variables in contravariant and covariant position are equalized.

```
class TPHsAndGenerics<UD> {

    Fun1$$<UD, UD> id = x -> x;

    <V extends UD> UD id2(V x) {
        return id.apply(x);}

    <AD extends AE, AB, AE>
    AB m(AB a, AD b){
        AE c = m2(a,b);
        return a;}

    <AM, AI> AI m2(AM a, AI b){
        return b;}
}
```

Figure 10: The complete simplified class `TPHsAndGenerics`

Theorem 7.13. *Let a Java-TX class*

$$\begin{aligned}&\mathit{Class}(\,cl, \mathit{extends}(\,ty\,), \overline{f}, \mathit{Method}(\,m_1, \overline{v_1}, bl_1\,)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Method}(\,m_n, \overline{v_n}, bl_n\,)\,)\end{aligned}$$

be given. Furthermore, let

$$\begin{aligned}&\mathit{Class}(cl, \mathit{extends}(\,ty\,), \overline{F\ f},\\ &\qquad \mathit{Method}(\,R_1, m_1, \overline{v_1 : A_1}, bl_1\,)\\ &\qquad \ldots\\ &\qquad \mathit{Method}(\,R_n, m_n, \overline{v_n : A_n}, bl_n\,)\,)\end{aligned}$$

be the result of the tree traversing, where type placeholders are introduced and let (cs, σ) *be a result of the type unification of* cl *(Sec. 3.1). Finally, let* $cs' = \{ cs'_{cl}, cs'_{m_1}, \ldots, cs'_{m_n} \}$ *the family of generated generics and* $cs'' = \{ cs''_{cl}, cs''_{m_1}, \ldots, cs''_{m_n} \}$ *the result family after applying Algorithm 7.7, eliminating inner type variables and equalizing type variables in contravariant and covariant position.*

Then

$$\begin{aligned}&\mathbf{Class}(\, cl, \mathbf{cs''_{cl}}, \textit{extends}(\, ty\,), \overline{\sigma(\, F\,)\ f}, \\ &\quad \textit{Method}(\, \mathbf{cs''_{m_1}}, \sigma(\, R_1\,), m_1, \overline{v_1 : \sigma(\, A_1\,)}, bl_1\,) \\ &\quad \ldots \\ &\quad \textit{Method}(\, \mathbf{cs''_{m_n}}, \sigma(\, R_n\,), m_n, \overline{v_n : \sigma(\, A_n\,)}, bl_n\,)\,)\end{aligned}$$

is a type correct Java program.

Proof. The constraints which are constructed during the tree traversing are the type conditions that corresponds to the type conditions of Java specification [10]. In [26] we proved that the type unification solves the constraints and the algorithm is sound and complete. This means the Java program is type correct if the instantiations for the remaining constraints cs, which consists only of type placeholders, are fulfilled.

Lemma 7.8 and Lemma 7.12 proves that Algorithm 7.7 and equalizing type variables in contravariant and covariant position, respectively, preserves the constraints. Therefore the instantiations of the corresponding members of cs'' into the Java program leads to a type correct Java program. □

8 Heterogeneous translation of function types

The actual Oracle's Java compiler implementations translate generics in a homogenous way. This means that all type parameters are erased. This property is called the *type erasure*. The main reason for the type erasure is that the heterogeneous translation (preserving the type parameters) would request for any type parameter instantiation that its own class be loaded with corresponding type parameters. This would slow down the runtime.

On the other hand, the type erasure leads to many disadvantages:

instanceof: Calling the `instanceof` operator with a parametrized type as second argument is not allowed.

Generic Exception: Classes that inherit from `Throwable` could not have parameters.

Generic overloading: Overloading a method with arguments with different parameter instantiations of the same class is not allowed.

In [20] for PIZZA a homogenous and a heterogeneous translation was described for a Java-like language. Furthermore, C# uses heterogeneous translation [14].

In the final version of the concept paper for the introduction of lambda expressions in Java [9] Brian Goetz offers arguments as to why they left out real function types: *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)`.

As we introduced with the special functional interfaces `Fun`N`$$` real function types, so we introduced into Java-TX heterogeneous translation for these functional interfaces.

In order to gain an understanding of the problem, let us consider the following example:

Example 8.1. *Let the following Java-TX program be given:*

```
class OLFun {
    m(f, x) {
      x = f.apply(x+x);
      return x;
    }
}
```

For the arguments of `m`*, we get the following typings:*

`(Fun1$$<Double, Double>,Double)`→`Double`
&
`(Fun1$$<Integer,Integer>,Integer)`→`Integer`
&
`(Fun1$$<String, String>,String)` → `String`.

*In Java bytecode the type parameters of the generic types are not considered (*type erasure*). Indeed, in bytecode the type parameters are contained in the signatures, but they are not included in the descriptors. The descriptors are used from the JVM during runtime. Therefore they are not used for resolving overloading. They are only*

used for the typecheck. The JVM considers only those descriptions where the type parameters are erased. For the class `OLFun` *the method headers in bytecode look like this:*

```
Double m(Fun1$$<Double,Double>,Double);
   descriptor: (LFun1$$;Double;)Double;
Integer m(Fun1$$<Integer,Integer>,Integer);
   descriptor: (LFun1$$;Integer;)Integer;
String m(Fun1$$<String,String>,String);
   descriptor: (LFun1$$;String;)String;
```

As in Example 4.1, this overloading is no problem either as the method call can be resolved by the second argument.

But if we erase the second argument

```
class OLFun {

  m(f) {
    var x;
    x = f.apply(x+x);
    return x;
  }
}
```

then the type erasure becomes a problem as the method headers in bytecode are:

```
Double m(Fun1$$<Double,Double>);
   descriptor: (LFun1$$;)Double;
Integer m(Fun1$$<Integer,Integer>);
   descriptor: (LFun1$$;)Integer;
String m(Fun1$$<String,String>);
   descriptor: (LFun1$$;)String;
```

Now method resolving is no longer possible as all three methods have the same argument `Fun1$$`*.*

This problem could only be solved by heterogeneous translations, which preserve the arguments in the descriptors. Unfortunately, the symbol < is not allowed in the descriptors. Therefore, following [20], we translate a type

$$\texttt{Fun}N\texttt{\$\$<}ty_1,\ldots,ty_n,ty_0\texttt{>}$$

to a string

$$\texttt{Fun}N\texttt{\$\$\$\_\$}ty_1\texttt{\$\_\$}\ldots\texttt{\$\_\$}ty_n\texttt{\$\_\$}ty_0\texttt{\$\_\$}$$

where ty_i are the descriptors of the type parameters. Descriptors are used to support the *type erasure* of any type and to differ classes with same names in different packages. The descriptors are subjected to the following substitutions:

1. . → /
2. / → $
3. ; → $_$

These substitutions are essential, so that the class name satisfies the Java specifications.

Differing from [20] we leave the class loader unchanged and for each used type

$$\texttt{Fun}N\texttt{\$\$\$\_\$}ty_1\texttt{\$\_\$}\ldots\texttt{\$\_\$}ty_n\texttt{\$\_\$}ty_0\texttt{\$\_\$}$$

we implement empty interface-files that inherit from

$$\texttt{Fun}N\texttt{\$\$<}ty_1,\ldots,ty_n,ty_0\texttt{>}:$$

```
interface FunN$$$_$ty1$_$...$_$tyN$_$ty0$_$
extends FunN$$<ty1,..., tyN,ty0> { }
```

Example 8.2. *For the class* `OLFun` *in Example 8.1 the following cutout of the bytecode is generated:*

```
Double m(Fun1$$<Double,Double>);
   descriptor:
(LFun1$$$_$Double$_$Double$_$;)Double;
Integer m(Fun1$$<Integer,Integer>);
   descriptor:
(LFun1$$$_$Integer$_$Integer$_$;)Integer;
String m(Fun1$$<String,String>);
   descriptor:
(LFun1$$$_$String$_$String$_$;)String;
```

Note that the prefix of the primitive types (`Ljava$lang$`) was left out for the sake of readability.

9 The inferred principal type

The principal types for the fields and the methods of a class are determined as given in Def. 5.2. The intersections (overloading) of the methods are derived from the elements of the result of the type unification $\{(\{\overline{T_1 \lessdot T_1'}\},\sigma_1),\ \ldots,\ (\{\overline{T_n \lessdot T_n'}\},\sigma_n)\}$ and the bound generics are derived as given in theorem 7.13.

As each σ_i is a general unifier (Theorem 1, [26]) and as during the adaptions of the remaining constraints $\{\overline{T_i \lessdot T_i'}\}$ either no solution is lost (building of family of generated generics, eliminating inner type variables, and equalizing type variables in contravariant and covariant position) or the reductions are indispensable to get a correct Java program (remove cycle and eliminate infima) the determined type is principal.

10 Related Work

Some object-oriented languages like Scala, C#, and Java perform *local* type inference [22, 24]. Local type inference means that missing type annotations are recovered using only information from adjacent nodes in the syntax tree without long distance constraints. For instance, the type of a variable initialized with a non-functional expression or the return type of a method can be inferred. However, method argument types, in particular for recursive methods, cannot be inferred by local type inference.

Milner's algorithm $\mathcal{W}$ [6, 18] is the gold standard for global type inference for languages with parametric polymorphism, which is used by ML-style languages. The fundamental idea of the algorithm is to enforce type equality by many-sorted type unification [17, 31]. This approach is effective and results in so-called principal types because many-sorted unification is unitary, which means that there is at most one most general result.

The presence of subtyping means that type unification is no longer unitary, but still finitary. Thus, there is no longer a single most general type, but any type is an instance of a finite set of maximal types. We have given an algorithm for Java type unification in [26] and proved soundness and completeness.

PIZZA [21] contained real function types with invariant arguments (no subtyping). Function types similar as in Java-TX are contained in Scala [23]. As shown in [29] our approach preserves the properties of target typing, while Scala do not have these properties.

There are different approaches of formal models of Java. In [13] a formal model of Java is given, where method-bodies consists of expressions rather than statements. There are two formal models. The first one for Java without generics and without lambda expressions. The second for GJ [5] with generics. In [34] wildcards are added. Finally, in [2] lambda expressions with functional interfaces are added.

In [32] we presented an extension of the calculus in [13] with type inference. This could be considered as a theorectical base of Java-TX without wildcards and without lambda expressions.

Ancona, Damiani, Drossopoulou, and Zucca [1] consider polymorphic byte code. Their approach is modular in the sense that it infers polymorphic structural types. As Java does not support structural types, their approach would have to be simulated with generated interfaces. In [28] we followed this approach. Furthermore Ancona and coworkers do not consider generic classes.

11 Implementation

A prototypical compiler for Java-TX has been implemented. The compiler is written in Java, itself. Additionally, an eclipse plugin is offered, presenting the full convenience of Java type inference. The official website is accessible with the following URL: https://www.hb.dhbw-stuttgart.de/javatx .

12 Summary and Outlook

Within the last 15 years Java has been developed so as to introduce various concepts from functional programming languages. In this paper we presented an extension of Java, called Java-TX. Java-TX continues the range of incorporating functional programming language features into Java. We added the feature of global type inference. Global type inference means that Java programs can be written without any type annotation. Java-TX preserves static typing through a type inference algorithm which infers a principal type for all fields and methods.

Subsequently, we showed how global type inference allows the extension of the overloading mechanism such that not only method identifiers but also, complete method declaration can be overloaded.

We discussed the principal type property in Java-TX and showed that a principal type of methods is an intersection of function types and that the type inference algorithm approximately infers the principal types. There are two restrictions: the polymorphic recursion and F-bounded polymorphism in combination with bounded wildcards.

Another extension of Java-TX is the introduction of real function types. We offered a number of examples which show that the lack of real function types is very harmful. We introduced Scala-like function types. For lambda expressions we defined these function types as explicit types. At the same time we preserved the concept of target typing for functional interfaces as was proposed in the so-called strawman approach in order to introduce lambda expressions into Java.

Additionally, we presented a concept for generalization for free type variables (generated generics) which is more powerful than in functional programming languages. The remaining type variables constraints of the type inference were distributed to the class and its method, respectively.

Finally, we showed that the overloading mechanism has some restrictions, caused by the *type erasure* (erasure of argument types in parametrized types during compilation). Therefore, in Java-TX we achieved an approach for heterogeneous translation of function types.

In the future, we plan to introduce a heterogeneous translation for all parametrized types. The following example in Fig. 11 shows that type inference suggests this.

```
import java.util.Vector;
import java.lang.Integer;
import java.lang.String;

class VectorAdd {
  vectorAdd(v1, v2) {
    var i = 0;
    var erg = new Vector<>();
    while (i < v1.size()) {
      erg.addElement(v1.elementAt(i)
                   + v2.elementAt(i));
      i++;
    }
    return erg;
  }
}
```

Figure 11: Heterogeneous translation

For the method `vectorAdd` the type

```
Vector<Integer>× Vector<Integer>
                          →Vector<Integer>
&
Vector<String>× Vector<String>
                          →Vector<String>
```

is inferred.

But as the type erasure erases the type parameters `Integer` and `String`, respectively, the overloading of the method `vectorAdd` cannot be resolved.

We shall consider two possible approaches for solving this problem. On the one hand, we could follow the PIZZA approach [20] to change the class loader. On the other hand, we could follow the ideas of [35] in a way similar to what we did for the function types. In [35] the types with instantiated parameters are subtypes of the non-instantiated ones. This approach could lead to a type hierarchy with multiple inheritance which is prohibited in Java.

Another feature derived from functional programming languages which has been introduced into Java in a restricted version is pattern matching. Pattern matching for the `instanceof` operator was introduced in Java 13-16 [8], for the `switch-case` instruction in Java 17-19 [4], and for the new record classes in Java 19 [3]. In PIZZA [21], pattern matching is realized via algebraic data types for the switch-case statement.

In combination with type inference, an approach of pattern matching in method headers similar to that of Haskell could be possible.

References

[1] Davide Ancona et al. “Polymorphic Bytecode: Compositional Compilation for Java-like Languages”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: ACM, 2005, pp. 26–37. ISBN: 1-58113-830-X. URL: http://doi.acm.org/10.1145/1040305.1040308.

[2] Lorenzo Bettini et al. “Java & Lambda: A Featherweight Story”. In: *Logical Methods in Computer Science* 14(3:17) (2018), pp. 1–24.

[3] Gavin Bierman. *JEP 405: Record Patterns (Preview)*. Updated: 2022/05/24 19:29. 2022. URL: http://openjdk.java.net/jeps/405.

[4] Gavin Bierman. *JEP 427: Pattern Matching for switch (Third Preview)*. Updated: 2022/05/25 16:51. 2022. URL: http://openjdk.java.net/jeps/427.

[5] Gilad Bracha et al. “Making the Future Safe for the Past: Adding Genericity to the Java Programming Language”. In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’98. Vancouver, British Columbia, Canada: ACM, 1998, pp. 183–200. ISBN: 1-58113-005-8. URL: http://doi.acm.org/10.1145/286936.286957.

[6] Luis Damas and Robin Milner. “Principal type-schemes for functional programs”. In: *Proc. 9th Symposium on Principles of Programming Languages* (1982).

[7] Brian Goetz. *JEP 286: Local-Variable Type Inference*. Updated: 2018/10/12 01:28. 2016. URL: http://openjdk.java.net/jeps/286.

[8] Brian Goetz. *JEP 394: Pattern Matching for instanceof*. Updated: 2021/03/01 16:07. 2020. URL: http://openjdk.java.net/jeps/394.

[9] Brian Goetz. *State of the Lambda*. Sept. 2013. URL: http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html.

[10] James Gosling et al. *The Java® Language Specification*. Java SE 8. The Java series. Addison-Wesley, 2014.

[11] Radu Grigore. “Java generics are turing complete”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 73–85. URL: http://dl.acm.org/citation.cfm?id=3009871.

[12] Fritz Henglein. “Type Inference with Polymorphic Recursion”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)*. Vol. 15(2). Apr. 1993, pp. 253–289.

[13] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. “Featherweight Java: a minimal core calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3 (2001), pp. 396–450.

[14] Andrew Kennedy and Don Syme. “Design and Implementation of Generics for the .NET Common Language Runtime”. In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI ’01. Snowbird, Utah, USA: Association for Computing Machinery, 2001, pp. 1–12. ISBN: 1581134142. URL: https://doi.org/10.1145/378795.378797.

[15] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. “The undecidablity of the semi-unification problem”. In: *Proceedings 22nd Annual ACM Symposium on Theory of Computation (STOC)*. Baltimore, Maryland, May 1990, pp. 468–476.

[16] Lambda. *Project Lambda: Java Language Specification draft*. Version 0.1.5. 2010. URL: http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt.

[17] A. Martelli and U. Montanari. “An Efficient Unification Algorithm”. In: *ACM Transactions on Programming Languages and Systems* 4 (1982), pp. 258–282.

[18] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–378.

[19] A. Mycroft. “Polymorphic type schemes and recursive definitions”. In: *Proc. 6th Int. Conf. on Programming*. Vol. LNCS 167. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 217–228. ISBN: 978-3-540-38809-8.

[20] Martin Odersky, Enno Runne, and Philip Wadler. “Two Ways to Bake Your Pizza – Translating Parameterised Types into Java”. In: *Proceedings of a Dagstuhl Seminar, Springer Lecture Notes in Computer Science* 1766 (2000), pp. 114–132.

[21] Martin Odersky and Philip Wadler. “Pizza into Java: Translating Theory into Practice”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’97. Paris, France: ACM, 1997, pp. 146–159. ISBN: 0-89791-853-3. URL: http://doi.acm.org/10.1145/263699.263715.

[22] Martin Odersky, Matthias Zenger, and Christoph Zenger. “Colored local type inference”. In: *Proc. 28th ACM Symposium on Principles of Programming Languages* 36.3 (2001), pp. 41–53.

[23] Martin Odersky et al. *The Scala Language Specification*. Version 2.13. 2019. URL: http://www.scala-lang.org/files/archive/spec/2.13.

[24] Benjamin C. Pierce and David N. Turner. “Local type inference”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’98. San Diego, California, United States, 1998, pp. 252–265.

[25] Martin Plümicke. “Intersection Types in Java”. In: *6th International Conference on Principles and Practices of Programming in Java*. Ed. by Luís Veiga et al. Vol. 347. ACM

International Conference Proceeding Series. Sept. 2008, pp. 181–188.

[26] Martin Plümicke. “Java type unification with wildcards”. In: *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*. Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf. Vol. 5437. Lecture Notes in Artificial Intelligence. Springer-Verlag Heidelberg, 2009, pp. 223–240.

[27] Martin Plümicke. “More Type Inference in Java 8”. In: *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. Ed. by Andrei Voronkov and Irina Virbitskaite. Vol. 8974. Lecture Notes in Computer Science. Springer, 2015, pp. 248–256.

[28] Martin Plümicke. “Structural Type Inference in Java-like Languages”. In: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016.* 2016, pp. 109–113. URL: http://ceur-ws.org/Vol-1559/paper09.pdf.

[29] Martin Plümicke and Andreas Stadelmeier. “Introducing Scala-like Function Types into Java-TX”. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: ACM, 2017, pp. 23–34. ISBN: 978-1-4503-5340-3. URL: http://doi.acm.org/10.1145/3132190.3132203.

[30] Mark Reinhold. *Project Lambda: Straw-Man Proposal*. Dec. 2009. URL: http://cr.openjdk.java.net/~mr/lambda/straw-man.

[31] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *Journal of ACM* 12(1) (Jan. 1965), pp. 23–41.

[32] Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. “Global Type Inference for Featherweight Generic Java”. In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 28:1–28:27. ISBN: 978-3-95977-225-9. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16256.

[33] Florian Steurer and Martin Plümicke. “Erweiterung und Neuimplementierung der Java Typunifikation”. In: *Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts*. Ed. by Jens Knoop, Martin Steffen, and Baltasar Trancón y Widemann. Research Report 482. ISBN 978-82-7368-447-9, (in german). Faculty of Mathematics and Natural Sciences, UNIVERSITY OF OSLO. 2018, pp. 134–149.

[34] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. “Wild FJ”. In: *Proceedings of FOOL 12*. Ed. by Philip Wadler. ACM. Long Beach, California, USA: School of Informatics, University of Edinburgh, Jan. 2005. URL: http://homepages.inf.ed.ac.uk/wadler/fool/.

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

IMPRESSUM

Schriftenreihe INSIGHTS
Themenreihe Engineering INSIGHTS

Herausgeber:
Fakultät Technik der
Dualen Hochschule Baden-Württemberg Stuttgart
Postfach 10 05 63, 70004 Stuttgart

Prof. Dr.-Ing. Harald Mandel, Prorektor und Dekan der Fakultät Technik
Jägerstraße 56, 70174 Stuttgart

E-Mail: harald.mandel@dhbw-stuttgart.de
Tel.: +49 711 1849 605
Fax: +49 711 1849 719

www.dhbw-stuttgart.de/technik/insights

Umschlaggestaltung: Kerstin Faißt
Bildnachweis: Gerd Altmann auf Pixabay bearbeitet von Kerstin Faißt
ISSN 2193-9098

ISSN 2193-9098

www.dhbw-stuttgart.de/technik/insights