Complete type inference in Java 8

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

Abstract

Java will be extended in version eight by closures and functional interfaces, whereupon functional interfaces are interfaces with one method. Functional interfaces represent the types of closures, which are also called lambda expressions. The type inference mechanism will be extended, such that the types of the parameters of lambda expressions could be inferred. But types of complete lambda expressions will still not be inferable. In this paper we give a type inference algorithm for complete lambda expressions and for methods. This means that fields, local variables, as well as parameters and return types of methods must not be typed explicitly.
We therefore define for a core of Java 8 an abstract syntax, we formalize the functional interfaces and define a type system for expressions and statements. Finally we give the type inference algorithm and prove its correctness and completeness.

Categories and Subject Descriptors D.1.5 [*Programming techniques*]: Object-oriented programming; D.2.2 [*Software engineering*]: Design tools and techniques—modules and interfaces; D.3.3 [*Programming languages*]: Language constructs and features—data types and structures

General Terms Algorithms, Theory

Keywords Code generation, language design, program design and implementation, type inference, type system

1. Introduction

In the Project lambda[1] a new version (version 8) of Java has been developed. The most important goal is to introduce programming patterns that allow modeling code such as data [7]. The principal includes the new features *lambda expressions*, *functional interfaces as target types*, *method and constructor references* and *default methods*. An essential enhancement is the introduction of lambda expressions. With the example which is also described in [7], we want to show the intention of our paper. 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

PPPJ '12, September, 12-14, 2012. Grahamstown, South Africa

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()));
```

But the type of the complete lambda expression must be known. In this case the type is given as the argument type of `sort` `Comparator<Person>`. 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].
The main purpose of our paper is to give a type inference algorithm which determine compatible target types for lambda expressions.

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 of an existing class or object whose typing is compatible with the corresponding functional type.
Our type inference algorithm is able to infer for method references the corresponding compatible target types, too.
Besides the type inference of the Java 8 extensions our algorithm infers the types of Java methods. This means that overloading and overriding must be considered, too.

[2] In earlier publications (e.g. [1]) functional interfaces are called SAM-types.

$$
\begin{array}{lll}
\mathit{Source} & := & \mathit{class}* \\
\mathit{class} & := & \mathsf{Class}(\mathit{type}, [\ \mathsf{extends}(\ \mathit{type}\ ),]\mathit{IVarDecl}*, \mathit{MethodDecl}*) \\
\mathit{FieldDecl} & := & \mathsf{Field}(\ [\underline{\boldsymbol{type}},]\mathit{var}[, \mathit{expr}]\ ) \\
\mathit{MethodDecl} & := & \mathsf{Method}(\ [\underline{\boldsymbol{type}},]\mathit{mname}, (\mathit{var}[: \underline{\boldsymbol{type}}])*, \mathit{block}\ ) \\
\mathit{block} & := & \mathsf{Block}(\ \mathit{stmt}*\ ) \\
\mathit{stmt} & := & \mathit{block}\ \mid\ \mathsf{Return}(\ \mathit{expr}\ )\ \mid\ \mathsf{While}(\ \mathit{bexpr}, \mathit{stmt}\ )\ \mid\ \mathsf{LocalVarDecl}(\ [\underline{\boldsymbol{type}},]\mathit{var}\ )\ \mid \mathsf{If}(\ \mathit{bexpr}, \mathit{stmt}[, \mathit{stmt}]\ )\ \mid\ \mathit{stmtexpr} \\
\mathit{lambdaexpr} & := & \mathsf{Lambda}(\ ((\mathit{var}[: \underline{\boldsymbol{type}}]))*, (\mathit{stmt}\ \mid\ \mathit{expr})\ ) \\
\mathit{methodref} & := & \mathsf{MethodRefClass}(\ \mathit{type}, \mathit{Method}\ )\ \mid\ \mathsf{MethodRefObject}(\ \mathit{iexpr}, \mathit{Method}\ )\ \mid\ \mathsf{MethodRefNew}(\ \mathit{type}\ ) \\
\mathit{stmtexpr} & := & \mathsf{Assign}(\ \mathit{vexpr}, \mathit{expr}\ )\ \mid\ \mathsf{MethodCall}(\ \mathit{iexpr}, f, \mathit{expr}*\ )\ \mid\ \mathsf{New}(\ \mathit{type}, \mathit{expr}*\ ) \\
\mathit{vexpr} & := & \mathsf{LocalVar}(\ \mathit{var}\ )\ \mid\ \mathsf{InstVar}(\ \mathit{iexpr}, \mathit{var}\ ) \\
\mathit{iexpr} & = & \mathit{vexpr}\ \mid\ \mathit{stmtexpr}\ \mid\ \mathsf{Cast}(\ \mathit{type}, \mathit{iexpr}\ )\ \mid\ \mathsf{this}\ \mid\ \mathsf{super} \\
\mathit{expr} & := & \mathit{lambdaexpr}\ \mid\ \mathit{methodref}\ \mid\ \mathit{iexp}\ \mid\ \mathit{bexp}\ \mid\ \mathit{sexp}^3
\end{array}
$$

Figure 1. The abstract syntax of a core of Java 8

In summary our type inference algorithm allows to write Java 8 programs without any type annotation. They were inferred during the compilation.
In [15] we presented an earlier version of Java also extended by closures [6]. We called the language Java$_\lambda$. We gave also a type inference algorithm for Java$_\lambda$.
With respect to type inference there are three main differences between Java$_\lambda$ and Java 8. While in Java 8 functional interfaces are target types of lambda expressions, in Java$_\lambda$ real function types are the types of lambda expressions. This induces that subtyping of types of lambda expressions is completely different in Java 8 and Java$_\lambda$.
In Java 8 there is no 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.
Additionally in Java 8 method and constructor references are added.

The paper is structured as follows. In the next section we define the abstract syntax for a reduced language of Java 8 and present a formal definition for the inferred functional interfaces. In the third section we give the type inference rules and we consider the type inference algorithm. In the fourth section we consider related work. Finally we close with a summary and an outlook.

2. The language

2.1 Abstract representation

The language (Figure 1) is an abstract representation of a core of Java 8. It is an extension of our language in [13]. The additional features are the lambda expressions and the method references. A lambda expression is an anonymous function and consists of optionally typed variables and either a statement or an expression. Method references play the same role as lambda expressions. The function is given as a reference to an existing method. There are three different kinds of method references: static methods, respectively instance methods of an arbitrary object of a particular class (MethodRefClass), methods of a particular object (MethodRefObject) and constructor references (MethodRefNew).
The concrete syntax in this paper of the lambda expressions is oriented at [7].
The optional type annotations [***type***] are the types which can be inferred by our type inference algorithm. In original Java 8 argument types of the lambda expressions can already be inferred. Our contributions are the type inference of the types of fields, the types of methods and the types of local variables.
For the type inference the language is restricted such that a typeless declared method must not be overloaded and overridden respectively and it must not overrides another method.

2.2 Functional interfaces and target types

In this section we will present the extensions of the Java 8 type system in a formalized way, which is defined in [2, 7]. Then we will extend Java 8 such that a type inference system can be defined.
The set of Java types Ty_p in a Java 8 program p is given as in [8], Section 4.5 and in [13], Section 2[4].
For the type descriptions of methods the Java types are not expressive enough. Therefore function types are defined:

Definition 2.1 (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* $ty, ty_i \in Fty_p \ \ (ty_1, \ldots, ty_n) \rightarrow ty \in Fty_p$

Function types are only used for the description of methods. They are not used in Java 8 programs.

In Java 8 lambda expressions are not typed by function types as in Java$_\lambda$. Instead functional interfaces are used.

Definition 2.2 (Functional interface). *An interface I, which have only one method, is called a* functional interface*.*

Many common callback interfaces have this property, such as `Runnable` and `Comparator`.

Definition 2.3 (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*[5]

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

Example 2.4. *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); }
```

[3] *sexp* and *bexp* stands for simple and boolean expressions, which are expressions of the base types `int` and `boolean`, respectively.

[4] In [13, 15] we called the Java types *simple types* $\mathsf{SType}_{TS}(\ BTV\ )$.

[5] Exceptions are not considered in this paper.

The lambda expression

```
(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 2.5. *As in [2] showed the same lambda expressions can be compatible with different types in different contexts. E.g. the lambda expression*

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

is in the contexts

```
Callable<String> c = () -> "done";
PrivilegedAction<String> a = () -> "done";
```

compatible with `Callable<String>` *respectively* `Privileged-Action<String>`[6].

Remark. *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.*

Now we will extend Java 8 for type inference, such that each lambda expression has a unique type description, which will be inferred by the type inference algorithm.

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

Definition 2.6 (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 2.7. *The relation* $\sim_{fi}$ *is an equivalence relation.*

Example 2.8. *In Example 2.5* `Callable<T>` *and* `Privileged-Action<T>` *are equivalent.*

Finally 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 2.4. 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 2.9 (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 2.10 (Canonical representation). *For each functional interface there is an unique N, such that an instance of* Fun*N is an equivalent functional interface.*
This instance is called canonical representation *of the equivalence class of functional interfaces.*

[6] In [7] these type are called *target types*.

Example 2.11. *The canonical representation of the compatible types of the lambda expression* `() -> "done"` *from Example 2.5 is* `Fun0<String>`.

2.3 Example

We will close the section by an example. We gave similar examples also in [13] and [15] for the respective Java versions. With this example the development of the programming language Java can be demonstrated.

Example 2.12. *The following Java 8 program implements the multiplication of matrices.*

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

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

  Fun2<Matrix, Matrix,Matrix>
  mul = (Matrix m1, Matrix m2) -> {
    Matrix ret = new Matrix ();
      for(int i = 0; i < size(); i++) {
          Vector<Integer> v1 = m1.elementAt(i);
          Vector<Integer> v2 = new Vector<Integer> ();
          for (int j = 0; j < v1.size(); j++) {
            int erg = 0;
            for (int k = 0; k < v1.size(); k++) {
                erg = erg + v1.elementAt(k)
                    * (m2.elementAt(k)).elementAt(j);
            }
            v2.addElement(erg);
          }
        ret.addElement(v2);
      }
    return ret; }

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

`op` is a function defined by a lambda expression. First it takes a matrix resulting in a function. This function takes another function which has as arguments two matrices and returns another matrix. The function `op` applies the function to its object (`this`) and its first argument. The method `mul` is the ordinary matrix multiplication in closure representation. Finally, in `main` the function `op` of the matrix `m1` is applied to the matrix `m2` and the function `mul` of `m1`.

3. Type inference

The base of many type inference algorithms is the algorithm $\mathcal{W}$ which was presented by Damas and Milner [4]. The fundamental idea of the algorithm is the type determination by type term unification [16]. In [13] we presented a type inference algorithm for Java 5.0 which bases on $\mathcal{W}$ and our type unification algorithm for Java 5.0 types [14]. In [15] we presented a type inference algorithm for Java$_\lambda$ which bases on the type inference algorithm which was presented by Fuh and Mishra [5] for a λ–calculus with subtyping but without overloading. Our contribution in this paper is a new type inference algorithm for Java 8 including overloading.

First we give the type inference rules, which can by considered as a specification for the algorithm.

3.1 Type inference rules

The type inference rules define how to derive the types for identifiers, expressions and statements under given assumptions.

$$[\textbf{lambda}_{stmt}] \quad \frac{(O \cup \{\, x_i : \theta_i \,\}, \tau, \tau') \ \rhd_{Stmt} \ \ s : \theta}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{Lambda}(\, (x_1 \!:\! \theta_1, \ldots, x_N \!:\! \theta_N), s \,) : \gamma} \ \ Comp(\, \mathsf{Lambda}(\, (x_1 : \theta_1, \ldots, x_N : \theta_N), s \,), \gamma \,)$$

$$[\textbf{lambda}_{expr}] \quad \frac{(O \cup \{\, x_i : \theta_i \,\}, \tau, \tau') \ \rhd_{Expr} \ \ e : \theta}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{Lambda}(\, (x_1 \!:\! \theta_1, \ldots, x_N \!:\! \theta_N), e \,) : \gamma} \ \ Comp(\, \mathsf{Lambda}(\, (x_1 \!:\! \theta_1, \ldots, x_N \!:\! \theta_N), e \,), \gamma \,)$$

$$[\textbf{MethodRefClass}] \quad \frac{(O_{\overline{\tau}}, \tau, \tau') \ \rhd_{Id} \ \ m : ty}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{MethodRefClass}(\, \overline{\tau}, m \,) : \gamma} \ \ Comp(\, m : ty, \gamma \,)$$

$$[\textbf{MethodRefObject}] \quad \frac{(O, \tau, \tau') \ \rhd_{Expr} \ \ re : \overline{\tau}, \quad (O_{\overline{\tau}}, \tau, \tau') \ \rhd_{Id} \ \ m : ty}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{MethodRefObject}(\, re, m \,) : \gamma} \ \ Comp(\, m : ty, \gamma \,)$$

$$[\textbf{MethodRefNew}] \quad \frac{(O_{\theta}, \tau, \tau') \ \rhd_{Id} \ \ \texttt{<init>}_{\theta} : ty}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{MethodRefNew}(\, \theta \,) : \gamma} \ \ Comp(\, \texttt{<init>}_{\theta} : ty, \gamma \,)$$

$$[\textbf{Assign}] \quad \frac{(O, \tau, \tau') \ \rhd_{Expr} \ \ ve : \theta', (O, \tau, \tau') \ \rhd_{Expr} \ \ e : \theta}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{Assign}(\, ve, e \,) : \theta'} \ \ \theta \leq^* \theta'$$

$$[\textbf{MethodCall}] \quad \frac{\begin{array}{c}(O, \tau, \tau') \ \rhd_{Expr} \ \ re : \overline{\theta}, \ \forall 1 \leqslant i \leqslant n : \ (O, \tau, \tau') \ \rhd_{Expr} \ \ e_i : \theta_i, \\ (\theta'_1 \ldots \theta'_n, \theta) = \texttt{lub}(\, \overline{\theta}, f, (\theta_1, \ldots, \theta_n) \,)\end{array}}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{MethodCall}(\, re, f(\, e_1, \ldots, e_n \,) \,) : \theta}$$

$$[\textbf{New}] \quad \frac{\forall 1 \leqslant i \leqslant n : \ (O, \tau, \tau') \ \rhd_{Expr} \ \ e_i : \theta_i, \quad (\theta'_1 \ldots \theta'_n, \theta) = \texttt{lub}(\, \theta, \texttt{<init>}_{\theta}, (\theta_1, \ldots, \theta_n) \,)}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{New}(\, \theta, (e_1, \ldots, e_n) \,) : \theta}$$

$$[\textbf{LocalVar}] \quad \frac{(O_{\tau}, \tau, \tau') \ \rhd_{Id} \ \ v : \theta}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{LocalVar}(\, v \,) : \theta}$$

$$[\textbf{InstVar}] \quad \frac{(O, \tau, \tau') \ \rhd_{Expr} \ \ re : \overline{\tau}, \quad (O_{\overline{\tau}}, \tau, \tau') \ \rhd_{Id} \ \ v : \theta}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{InstVar}(\, re, v \,) : \theta}$$

$$[\textbf{Cast}] \quad \frac{(O, \tau, \tau') \ \rhd_{Expr} \ \ e : \overline{\theta}}{(O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{Cast}(\, \theta, e \,) : \theta}$$

$$[\textbf{This}] \quad (O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{this} : \tau$$

$$[\textbf{Super}] \quad (O, \tau, \tau') \ \rhd_{Expr} \ \ \mathsf{super} : \tau'$$

Figure 2. Expression rules

For the type inference system we need some additional definitions: A set of *type assumptions* O is a map indexed by class names, which maps method and variable names to types (e.g. $O_{\texttt{Matix}} = \{\, \texttt{mul} : \texttt{Fun2<Matrix, Matrix, Matrix>} \,\}$). The elements of the respective sets O_τ are determined by the class declarations, the inheritance and the visibility.

In the following σ denotes a *substitution*, which substitutes some type variables by types.

First we need an implication $\rhd_{Id}$. We write

$$(O_{\overline{\tau}}, \tau, \tau') \ \rhd_{Id} \ \ f : ty,$$

if in the class τ, whose direct superclass is τ', for an identifier f the type ty is derivable from the type assumptions of the class $\overline{\tau}$.

Additionally we need two implications $\rhd_{Expr}$ and $\rhd_{Stmt}$.

$(O, \tau, \tau') \ \rhd_{Expr} \ \ exp : \theta$ means that under the type assumptions O in the class τ, whose direct superclass is τ', the expression *exp* has the type θ.

$(O, \tau, \tau') \ \rhd_{Stmt} \ \ stmt : \theta$ means that under the type assumptions O in the class τ, whose direct superclass is τ' the statement *stmt* has the type θ.

3.1.1 Ident–rules

The **Ident**–rules defines the typings of identifiers. The **Ident**–rules differ, if an identifier is declared in the actual class τ or in another class. If an identifier is declared in the actual class τ all type variables must not be instantiated. Otherwise any instance is allowed[7].

[7] In [4] this differentiation is given by type schemes and types.

$$[\textbf{Return}]\quad \frac{(O,\tau,\tau')\ \rhd_{Expr}\ \ e:\theta}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Return}(\,e\,):\theta}$$

$$[\textbf{BlockInit}]\quad \frac{(O,\tau,\tau')\ \rhd_{Stmt}\ \ stmt:\theta}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Block}(\,stmt\,):\theta}$$

$$[\textbf{Block}]\quad \frac{(O,\tau,\tau')\ \rhd_{Stmt}\ \ s_1:\theta,\ \ (O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Block}(\,s_2;\ldots;s_n;\,):\theta'}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Block}(\,s_1;s_2;\ldots;s_n;\,):\overline{\theta}}\quad \overline{\theta}\in\textbf{MUB}(\,\theta,\theta'\,)$$

$$[\textbf{Blockvoid}]\quad \frac{(O,\tau,\tau')\ \rhd_{Stmt}\ \ s_1:\texttt{Void},\ \ (O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Block}(\,s_2;\ldots;s_n;\,):\theta}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Block}(\,s_1;s_2;\ldots;s_n;\,):\theta}$$

$$[\textbf{BlockLoVarDecl}]\quad \frac{\begin{array}{c}O'_\tau=O_\tau\backslash\{\,v:\theta'\,\}\cup\{\,v:\overline{\theta}\,\}\\ ((O\backslash O_\tau)\cup O'_\tau,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Block}(\,s_2;\ldots;s_n;\,):\theta\end{array}}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Block}(\,\mathsf{LocalVarDecl}(\,v,\overline{\theta}\,);s_2;\ldots;s_n;\,):\theta}$$

$$[\textbf{If}]\quad \frac{\begin{array}{c}(O,\tau,\tau')\ \rhd_{Stmt}\ \ s_1:\theta,(O,\tau,\tau')\ \rhd_{Stmt}\ \ s_2:\theta'\\ (O,\tau,\tau')\ \rhd_{Expr}\ \ e:\texttt{boolean}\end{array}}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{If}(\,e,s_1,s_2\,):\overline{\theta}}\quad \overline{\theta}\in\textbf{MUB}(\,\theta_1,\theta_2\,)$$

$$[\textbf{Assign}]\quad \frac{(O,\tau,\tau')\ \rhd_{Expr}\ \ \mathsf{Assign}(\,ve,e\,):\theta}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{Assign}(\,ve,e\,):\texttt{Void}}$$

$$[\textbf{New}]\quad \frac{(O,\tau,\tau')\ \rhd_{Expr}\ \ \mathsf{New}(\,\theta,(e_1,\ldots,e_n)\,):\theta}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{New}(\,\theta,(e_1,\ldots,e_n)\,):\texttt{Void}}$$

$$[\textbf{MethodCall}]\quad \frac{(O,\tau,\tau')\ \rhd_{Expr}\ \ \mathsf{MethodCall}(\,e,f(\,e_1,\ldots,e_n\,)\,):\theta}{(O,\tau,\tau')\ \rhd_{Stmt}\ \ \mathsf{MethodCall}(\,e,f(\,e_1,\ldots,e_n\,)\,):\texttt{Void}}$$

Figure 3. Statement Rules

$$[\textbf{Ident}]\quad \frac{(\,f:ty)\in O_\tau}{(O_\tau,\tau,\tau')\ \rhd_{Id}\ \ f:ty}$$

$$[\textbf{IdentGen}]\quad \frac{(\,f:ty)\in O_{\overline{\tau}}}{(O_{\overline{\tau}},\tau,\tau')\ \rhd_{Id}\ \ f:(\sigma'\circ\sigma)(\,ty\,)}\quad \overline{\tau}\neq\tau$$

3.1.2 Expression rules

In Figure 2 the type inference rules for the important Java 8 expressions are given.

There are two **lambda**–rules, as the body of the lambda expression can either be a statement or an expression. In both cases the types of the lambda expressions are functional interfaces, which are compatible with the corresponding lambda expression.

The **MethodRefClass/Object/New**–rules are similar to the rules for the lambda expressions. From the type of the respective method a corresponding functional interface is derived.

The **Assign**-rule is canonically defined.

In the **MethodCall**–rule first the type $\overline{\theta}$ of the receiver re is derived. Then the types θ_i of the arguments e_i are derived, which are subtypes of the argument types θ'_i of the method. The arity and the result type of the method $(\theta'_1,\ldots,\theta'_n,\theta)$ is the determined as the least upper bound arity of the method greater than $(\theta_1,\ldots,\theta_n)$ by:

Definition 3.1. (`lub` (least upper bound)) *The function* `lub` $: Ty_p \times \mathsf{Identifiers} \times {Ty_p}^* \rightarrow {Ty_p}^* \times Ty_p$ *is defined by:*

$$\texttt{lub}(\,\overline{\theta},f,\theta_1\ldots\theta_n\,)=(\theta'_1\ldots\theta'_n,\theta),$$

if $(\theta'_1\ldots\theta'_n)$ *is the smallest tuple with*

$(\theta_1,\ldots,\theta_n)\leq^*(\theta'_1,\ldots,\theta'_n)$ *and* $O_{\overline{\theta}}\rhd_{Id} f:(\theta'_1,\ldots,\theta'_n)\rightarrow\theta.$

In the **New**–rule `<init>`$_\theta$ denotes a constructor of the class θ. The rule is similar to the **MethodCall**–rule. Only the type for the receiver is not derived and the result type is given as θ.

The **LocalVar**–rule derives the type of variables of the actual method .

The **Cast**–rule casts the type of the given expression e to the given type θ.

The **InstVar**–rule types identifiers which are defined in a class $\overline{\theta}$ as fields.

The **This**–rule types the expression this by its class τ.

The **Super**–rule types the expression super by its superclass τ'.

For boolean expressions *bexp* and simple expressions *sexp* the rules are defined analogously.

3.1.3 Statement rules

In Fig. 3 the type inference rules for the important Java 8 statements are given.

The Return–statement determines the result type of a list of statements, which is closed by the return-statement. Therefore the statement gets the corresponding expression type. If there is no return-statement, there is no result type. We infer in this case `Void`[8].

The type of a block of statements is basically defined by the type of its closing statement (rule **BlockInit**). Stepwise the type of a block is given by the minimal upper bound (**MUB**) of the first statement and the type of the block of the rest-statements (**Block**-rule). If the type of the first statement is given as `Void` then the type of the block is preserved (**Blockvoid**-rule). The **BlockLoVarDeclBlock**–rule replaces the assumption of the declared typed variable in the set of type assumptions. As the type of the LocalVarDecl statement is `Void` the type of the block is unchanged.

The type of the If–statement is determined by the minimal upper bound of the type of if- and the else-branch.

The statements Assign, New and MethodCall have the type `Void`, as no result is returned.

3.2 Type inference algorithm

The type inference algorithm for Java 8 is a combination of our approaches in [13] for Java 5.0 and in [15] for Java$_\lambda$. We adapt the function **TYPE** from [15] by introducing the interfaces `Fun`*N* and overloading respective overriding for Java methods. We solve the resulting constraints[9] by the type unification of [14].

First we have to give two auxiliary definitions.

Definition 3.2 (Set of type assumptions `TypeAssumptions`). *The* set of type assumptions *contains three different forms of elements:*

$v : \theta$**:** *Assumptions for fields or local variables of the actual class.*
$\tau.v : \theta$**:** *Assumptions for fields of the class τ.*
$\tau.m : (\theta_1, \ldots, \theta_n) \rightarrow \theta$**:** *Assumptions for the methods of the class τ.*

Additionally we extend the set of constraints, such that for the alternative types of overloaded and overridden methods also constraints can be given.

Definition 3.3 (Set of constraints `ConstraintsSet`). *The* set of constraints *consists of constraints of the form $\theta\, R\, \theta'$, where θ and θ' are Java types and R ($R \in \{\lessdot, \lessdot_?, \doteq\}$)[10] is a subtyping condition. As method overloading and overriding are allowed two new symbols $\vee$ and $\|$ are introduced. These symbols stands for alternatives in the set of constraints. The $\vee$–symbol is used for constraints, which are deduced by overloaded methods and the $\|$–symbol is used for constraints, which are deduced by overridden methods.*

Both new symbols $\vee$ and $\|$ can be considered as disjunctions.

In this paper we will present all algorithms again in a functional style, like in Haskell. We use the let–construction and pattern matching, which means that for each data-constructor in functions an own equation is given.

3.2.1 The function TYPE

The function **TYPE** inserts type annotations, widely type variables as placeholders, in the Java class and determines a set of type constraints.

In **TYPE** the functions **TYPEExpr** and **TYPEStmt** determine the constraints for the expressions and the statements respectively.

[8] In [2] the inference of `Void` and the relation to the base type `void` is an unresolved question.

[9] In [15] the constraints were called coercions.

[10] In [14] we introduced besides the usual subtyping condition $\lessdot$ two other subtyping conditions $\lessdot_?$ and $\doteq$ for subterm subtyping.

The function **TYPE** is given as:

$$\textbf{TYPE}: \texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \texttt{TClass} \times \texttt{ConstraintsSet}$$

$$
\begin{array}{l}
\textbf{TYPE}(\, Ass, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}, \mathit{mdecls} \,)\,) = \textbf{let} \\
\quad \mathit{fdecls} = [\mathsf{Field}(\, f_1, lexpr_1 \,), \ldots, \mathsf{Field}(\, f_n, lexpr_n \,)]^{11} \\
\quad \mathit{mdecls} = [\mathsf{Method}(\, me_1, (v_1, \ldots, v_{m_1}), bl_1 \,), \ldots, \\
\qquad\qquad \mathsf{Method}(\, me_m, (v_1, \ldots, v_{m_m}), bl_m \,)]^{11} \\
\quad \mathit{ftypeass} = \{\, \texttt{this}.f_i : a_i \mid a_i \textit{ fresh type variables} \,\} \\
\qquad\qquad \cup \{\, \texttt{this}.m_j : (b_{j1}, \ldots, b_{jm_i}) \rightarrow b_j \\
\qquad\qquad\quad b_j, b_{jk} \textit{ fresh type variables} \\
\qquad\qquad \cup \{\, \texttt{this} : \tau, \texttt{super} : \tau' \,\} \\
\qquad\qquad \cup \{\, \textit{visible types of methods and fields of } \tau' \,\} \\
\quad \mathit{AssAll} = \mathit{Ass} \cup \mathit{ftypeass} \\
\quad \textbf{Forall } 1 \leqslant i \leqslant n \\
\qquad (lexp_{i_t} : rtyF_i, ConSF_i) = \textbf{TYPEExpr}(\, AssAll, lexpr_i \,) \\
\quad \textbf{Forall } 1 \leqslant j \leqslant m \\
\qquad (bl_{j_t} : rtyM_j, ConSM_j) = \textbf{TYPEStmt}(\, AssAll, bl_j \,) \\
\quad \mathit{fdecls}_t = \\
\qquad [\mathsf{Field}(\, a_1, f_1, lexpr_{1_t} \,), \ldots, \mathsf{Field}(\, a_n, f_n, lexpr_{n_t} \,)] \\
\quad \mathit{mdecls}_t = \\
\qquad [\mathsf{Method}(\, b_1, me_1, (v_1 : b_{11}), \ldots, (v_{m_1} : b_{1m_1}), bl_{1_t} \,) \\
\qquad , \ldots, \\
\qquad \mathsf{Method}(\, b_m, me_m, (v_1 : b_{m1}), \ldots, (v_{m_m} : b_{mm_m}), bl_{m_t} \,)] \\
\textbf{in} \\
\quad (\mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}_t, \mathit{mdecls}_t \,), \\
\quad\; (\bigcup_i ConSF_i \cup \{\, (rTyF_i \lessdot a_i) \mid 1 \leqslant i \leqslant n \,\}) \cup \\
\quad\; \bigcup_i ConSM_i \cup \{\, (rTyM_i \lessdot b_j) \mid 1 \leqslant j \leqslant m \,\})
\end{array}
$$

In the following type variables of identifier's types are refreshed, if there are not members of the actual class. This is done such that different instances of the type variables are possible[12]. The function **fresh** refreshes the type variables.
The function **ass**(`this`) gives the type assumption of the actual class (type assumption of `this`).

The function **TYPEExpr** is given as:

$$\textbf{TYPEExpr}: \texttt{TypeAssumptions} \times \texttt{Expr} \rightarrow \texttt{TExpr} \times \texttt{ConstraintsSet}$$

$$
\begin{array}{l}
\textbf{TYPEExpr}(\, Ass, \mathsf{Lambda}(\, (x_1, \ldots, x_N), expr | stmt \,)\,) = \\
\;\textbf{let} \\
\quad AssArgs = \{\, x_i : a_i \mid a_i \textit{ fresh type variables} \,\} \\
\quad (expr_t : rty, ConS) = \textbf{TYPEExpr}(\, Ass \cup AssArgs, expr \,) \\
\;\; | \, (stmt_t : rty, ConS) = \textbf{TYPEStmt}(\, Ass \cup AssArgs, stmt \,) \\
\;\textbf{in} \\
\quad (\mathsf{Lambda}(\, (x_1 : a_1, \ldots, x_N : a_N), expr_t : rty | stmt_t : rty \,) : a, \\
\quad\; ConS \cup \{\, (\texttt{Fun}N\texttt{<}rty, a_1, \ldots, a_N\texttt{>} \lessdot a) \,\}), \\
\quad \textit{where } a \textit{ is a fresh type variable}
\end{array}
$$

$$
\begin{array}{l}
\textbf{TYPEExpr}(\, Ass, \mathsf{MethodRefClass}(\, \tau, m \,)\,) = \\
\quad (\mathsf{MethodRefClass}(\, \tau, m \,) : a, \\
\quad \{\bigvee_{\tau.m:(\theta_1, \ldots, \theta_N) \rightarrow \theta \in Ass} \{\, (\texttt{Fun}N\texttt{<}\tilde{\theta}, \tilde{\theta}_1, \ldots, \tilde{\theta}_N\texttt{>} \lessdot a)\}\}), \\
\quad \textit{where } \tilde{\theta} = \theta \textit{ and } \tilde{\theta}_i = \theta_i, \textit{ if } \tau = \textbf{ass}(\, \texttt{this} \,), \\
\quad \textit{otherwise } \tilde{\theta} = \textbf{fresh}(\, \theta \,), \tilde{\theta}_i = \textbf{fresh}(\, \theta_i \,) \\
\quad \textit{and } a \textit{ is a fresh type variable}
\end{array}
$$

$$
\begin{array}{l}
\textbf{TYPEExpr}(\, Ass, \mathsf{MethodRefObject}(\, re, m \,)\,) = \\
\quad \textbf{let } (re_t : rty, ConS) = \textbf{TYPEExpr}(\, Ass, re \,) \\
\quad \textbf{in } (\mathsf{MethodRefObject}(\, re_t : rty, m \,) : a, \\
\qquad\quad ConS \cup \{\bigvee_{\tau.m:(\theta_1, \ldots, \theta_N) \rightarrow \theta \in Ass} \\
\qquad\qquad\qquad \{\, (rty \lessdot \tau), (\texttt{Fun}N\texttt{<}\tilde{\theta}, \tilde{\theta}_1, \ldots, \tilde{\theta}_N\texttt{>} \lessdot a) \,\}\,\}),
\end{array}
$$

[11] We assume without loss of generality that all fields and methods are declared typeless and that all fields are initialized by expressions.

[12] In [4] this differentiation is given by type schemes and types.

where $\tilde{\theta} = \theta$ *and* $\tilde{\theta}_i = \theta_i$, *if* $\tau = \textbf{ass}(\,\texttt{this}\,)$,
otherwise $\tilde{\theta} = \textbf{fresh}(\,\theta\,), \tilde{\theta}_i = \textbf{fresh}(\,\theta_i\,)$
and a *is a fresh type variable*

$\textbf{TYPEExpr}(\,Ass, \mathsf{MethodRefNew}(\,\theta\,)\,) =$
$(\mathsf{MethodRefNew}(\,\theta\,) : a,$
$\{\,\bigvee_{\theta.\texttt{<init>}_\theta:(\theta_1,\ldots,\theta_N)\,\rightarrow\,\theta\in Ass}$
$\{\,(\texttt{Fun}N\texttt{<}\tilde{\theta}, \tilde{\theta}_1, \ldots, \tilde{\theta}_n\texttt{>} \lessdot a)\,\}\,\}$,
where $\tilde{\theta} = \theta$ *and* $\tilde{\theta}_i = \theta_i$, *if* $\theta = \textbf{ass}(\,\texttt{this}\,)$,
otherwise $\tilde{\theta} = \textbf{fresh}(\,\theta\,), \tilde{\theta}_i = \textbf{fresh}(\,\theta_i\,)$
and a *is a fresh type variable*

$\textbf{TYPEExpr}(\,Ass, \mathsf{Assign}(\,ve, e\,)\,) =$
let
$(e_t : rty_2, ConS_2) = \textbf{TYPEExpr}(\,Ass, e\,)$
$(ve_t : rty_1, ConS_1) = \textbf{TYPEExpr}(\,Ass, ve\,)$
in
$(\mathsf{Assign}(\,ve_t : rty_1, e_t : rty_2\,) : a,$
$CoesS_1 \cup CoesS_2 \cup$
$\{\,(rty_2 \lessdot rty_1), (rty_1 \lessdot a)\,\})$,
where a *is a fresh type variable*

$\textbf{TYPEExpr}(\,Ass, \mathsf{MethodCall}(\,re, m(\,e_1, \ldots, e_n\,)\,)\,) =$
let
$(re_t : rty, ConS) = \textbf{TYPEExpr}(\,Ass, re\,)$
$(e_{i_t} : rty_i, ConS_i) = \textbf{TYPEExpr}(\,Ass, e_i\,), \forall 1 \leqslant i \leqslant n$
in
$(\mathsf{MethodCall}(\,re_t{:}rty, m(\,e_{1_t}{:}rty_1, \ldots, e_{n_t}{:}rty_n\,)\,){:}a,$
$ConS \cup \bigcup_i ConS_i \cup$
$\{\,\textbf{overloading}(\,m, Ass, (rty, (rty_1, \ldots, rty_n)), a\,)\,\}$
where a *is a fresh type variable*

where **overloading** is given as

$\textbf{overloading}(\,m, Ass, (\overline{\tau}, (\overline{\theta}_1, \ldots, \overline{\theta}_n), a)\,) =$
let
Ass_m = *set of all type assumptions for* m *with* n *arguments*
$MAss_m$ = *set of all type assumptions in* Ass_m *where the tupel (receiver, argtypes, rettype) is minimal wrt.* $\leq^*$.
in
$\bigvee_{ass\in MAss_m}(\textbf{constraints}(\,ass, (\overline{\tau}, (\overline{\theta}_1, \ldots, \overline{\theta}_n), a)\,) \;\|$
$\|_{ass'\in\textbf{sargs}(\,ass, Ass\,)}\,\textbf{constraints}(\,ass', (\overline{\tau}, (\overline{\theta}_1, \ldots, \overline{\theta}_n), a)\,)\,)$
with
$\textbf{constraints}(\,\texttt{this}.m : (\theta_1, \ldots, \theta_n) \rightarrow \theta, (\overline{\tau}, (\overline{\theta}_1, \ldots, \overline{\theta}_n), a)\,) =$
$\{\,\overline{\tau} \lessdot \textbf{ass}(\,\texttt{this}\,)\,\} \cup \{\,\overline{\theta}_i \lessdot \theta_i \mid 1 \leqslant i \leqslant n\,\} \cup \{\,\theta \lessdot a\,\}$
$\textbf{constraints}(\,\tau.m : (\theta_1, \ldots, \theta_n) \rightarrow \theta, (\overline{\tau}, (\overline{\theta}_1, \ldots, \overline{\theta}_n), a)\,) =$
let
$(\tilde{\tau}, (\tilde{\theta}_1, \ldots, \tilde{\theta}_n) \rightarrow \tilde{\theta}) = \textbf{fresh}(\,\tau, (\theta_1, \ldots, \theta_n) \rightarrow \theta\,)$
in
$\{\,\overline{\tau} \lessdot \tilde{\tau}\,\} \cup \{\,\overline{\theta}_i \lessdot \tilde{\theta}_i \mid 1 \leqslant i \leqslant n\,\} \cup \{\,\tilde{\theta} \lessdot a\,\}$
and
$\textbf{sargs}(\,\tau.m : (\theta_1, \ldots, \theta_n) \rightarrow \theta, Ass\,) =$
$\{\,\tau.m : (\theta'_1, \ldots, \theta'_n) \rightarrow \theta'' \in Ass \mid \theta_i \lneq^* \theta'_i, 1 \leqslant i \leqslant n\,\}$

overloading determines for all possible overloadings and overridings of a method the constraints, where **constraints** itself forms the constraints from the receiver type, the argument types, the return type and a given type assumption for the method. If it is a method from a class, which is not the actual class (`this`), all type variables are replaced by fresh type variables (**fresh**), as different instances can occur. **sargs** determines all type assumptions of a method, where the argument types are supertypes of a minimal type assumption.

We give a small example for **overloading**:

Example 3.4. *Let for the method* `m` *a set of type assumptions*

$Ass_m = \{\,\texttt{A.m} : \texttt{String} \rightarrow \texttt{String},$
$\texttt{A.m} : \texttt{Integer} \rightarrow \texttt{Integer},$
$\texttt{A.m} : \texttt{Object} \rightarrow \texttt{Object},$
$\texttt{A}'\texttt{.m} : \texttt{Object} \rightarrow \texttt{Object},$
$\texttt{B.m} : \texttt{Float} \rightarrow \texttt{Float}\,\}$

be given, with $\texttt{A} \leq^* \texttt{A}'$, $\texttt{B} \not\leq^* \texttt{A}$ *and* $\texttt{A} \not\leq^* \texttt{B}$. *Then holds:*

$MAss_m = \{\,\texttt{A.m} : \texttt{String} \rightarrow \texttt{String},$
$\texttt{A.m} : \texttt{Integer} \rightarrow \texttt{Integer},$
$\texttt{A}'\texttt{.m} : \texttt{Object} \rightarrow \texttt{Object},$
$\texttt{B.m} : \texttt{Float} \rightarrow \texttt{Float}\,\}$

$\textbf{overloading}(\,m, Ass_m, (\overline{\tau}, (\overline{\theta}_1), a)\,) =$
$\{\,\overline{\tau} \lessdot \texttt{A}, \overline{\theta}_1 \lessdot \texttt{String}, \texttt{String} \lessdot a\,\}$
$\|\;\{\,\overline{\tau} \lessdot \texttt{A}, \overline{\theta}_1 \lessdot \texttt{Object}, \texttt{Object} \lessdot a\,\}$
$\vee\{\,\overline{\tau} \lessdot \texttt{A}, \overline{\theta}_1 \lessdot \texttt{Integer}, \texttt{Integer} \lessdot a\,\}$
$\|\;\{\,\overline{\tau} \lessdot \texttt{A}, \overline{\theta}_1 \lessdot \texttt{Object}, \texttt{Object} \lessdot a\,\}$
$\vee\{\,\overline{\tau} \lessdot \texttt{A}', \overline{\theta}_1 \lessdot \texttt{Object}, \texttt{Object} \lessdot a\,\}$
$\vee\{\,\overline{\tau} \lessdot \texttt{B}, \overline{\theta}_1 \lessdot \texttt{Float}, \texttt{Float} \lessdot a\,\}$

Let us continue the function **TYPEExpr**.

$\textbf{TYPEExpr}(\,Ass, \mathsf{New}(\theta, e_1, \ldots, e_n\,)\,) =$
let
$(e_{i_t} : rty_i, ConS_i) = \textbf{TYPEExpr}(\,Ass, e_i\,), \forall 1 \leqslant i \leqslant n$
in
$(\mathsf{New}(\,\theta, e_{1_t} : rty_1, \ldots, e_{n_t} : rty_n\,) : a,$
$\bigcup_i ConS_i \cup$
$\{\,\textbf{overloading}(\,\texttt{<init>}_\theta, Ass, (\theta, (rty_1, \ldots, rty_n)), a\,)\,\}$

$\textbf{TYPEExpr}(\,Ass, \mathsf{InstVar}(\,re, v\,)\,) =$
let
$(rty, ConS) = \textbf{TYPEExpr}(\,Ass, re\,)$
in
$(\mathsf{InstVar}(\,re : rty, v\,) : a,$
$ConS \cup \{\,\bigvee_{\tau.v:\theta\in Ass}\{\,(rty \lessdot \tau), (\tilde{\theta} \lessdot a)\,\}\,\}$
where $\tilde{\theta} = \theta$, *if* $\tau = \textbf{ass}(\,\texttt{this}\,)$, *otherwise* $\tilde{\theta} = \textbf{fresh}(\,\theta\,)$
and a *is a fresh type variable*

We omit the remaining cases of **TYPEExpr** for LocalVar, and Cast. These are given analogously.

In the functions **TYPE** and **TYPEExpr** the function **TYPEStmt** for the typing of statements is called. The function **TYPEStmt** is given as:

TYPEStmt: $\texttt{TypeAssumptions} \times \texttt{Stmt}$
$\rightarrow \texttt{TStmt} \times \texttt{ConstraintsSet}$

$\textbf{TYPEStmt}(\,Ass, \mathsf{Return}(\,e\,)\,) =$
let
$(e_t : rty, ConS) = \textbf{TYPEExpr}(\,Ass, e\,)$
in
$(\mathsf{Return}(\,e_t : rty\,) : a, ConS \cup \{\,(rty \lessdot a)\,\})$
where a *is a fresh type variable*

$\textbf{TYPEStmt}(\,Ass, \mathsf{Block}(\,s\,)\,) =$
let
$(s_t : rty, ConS) = \textbf{TYPEStmt}(\,Ass, s\,)$
in
$(\mathsf{Block}(\,s_t : rty\,) : rty, ConS)$

$$
\begin{array}{l}
\textbf{TYPEStmt}(\,Ass, \mathsf{Block}(\,\mathsf{LocalVarDecl}(\,v, \overline{\theta}\,), s_2, \ldots, s_n\,)\,) = \\
\quad \textbf{let} \\
\qquad (\mathsf{Block}(\,s_{2_t}, \ldots, s_{n_t}\,) : rty, ConS) = \\
\qquad\quad \textbf{TYPEStmt}(Ass \backslash \{\, v : \theta' \,\} \cup \{\, v : \overline{\theta} \,\}, \\
\qquad\qquad\qquad \mathsf{Block}(\,s_2, \ldots, s_n\,)\,) \\
\quad \textbf{in} \\
\qquad (\mathsf{Block}(\,\mathsf{LocalVarDecl}(\,v, \overline{\theta}\,) : \mathtt{Void}, s_{2_t}, \ldots, s_{n_t}\,) : rty, \\
\qquad\; ConS)
\end{array}
$$

$$
\begin{array}{l}
\textbf{TYPEStmt}(\,Ass, \mathsf{Block}(\,s_1, \ldots, s_n\,)\,) = \\
\quad \textbf{let} \\
\qquad (s_{1_t} : rty_1, ConS_1) = \textbf{TYPEStmt}(\,Ass, s_1\,) \\
\qquad (\mathsf{Block}(\,s_{2_t}, \ldots, s_{n_t}\,) : rty_2, ConS_2) = \\
\qquad\quad \textbf{TYPEStmt}(\,Ass, \mathsf{Block}(\,s_2, \ldots, s_n\,)\,) \\
\quad \textbf{in} \\
\qquad (\mathsf{Block}(\,s_{1_t} : rty_1, s_{2_t}, \ldots, s_{n_t}\,) : a, \\
\qquad\; ConS_1 \cup ConS_2 \cup \{\, (rty_1 \lessdot a), (rty_2 \lessdot a) \,\}) \\
\qquad \textit{where } a \textit{ is a fresh type variable}
\end{array}
$$

$$
\begin{array}{l}
\textbf{TYPEStmt}(\,Ass, \mathsf{if}(\,e, s_1, s_2\,)\,) = \\
\quad \textbf{let} \\
\qquad (e_t : rty_0, ConS_0) = \textbf{TYPEExpr}(\,Ass, e\,) \\
\qquad (s_{1_t} : rty_1, ConS_1) = \textbf{TYPEStmt}(\,Ass, s_1\,) \\
\qquad (s_{2_t} : rty_2, ConS_2) = \textbf{TYPEStmt}(\,Ass, s_2\,) \\
\quad \textbf{in} \\
\qquad (\mathsf{if}(\,e_t : rty_0, s_{1_t} : rty_1, s_{2_t} : rty_2\,) : a, \\
\qquad\; ConS_0 \cup ConS_1 \cup ConS_2 \cup \\
\qquad\quad \{\, (rty_0 \lessdot \mathtt{boolean}), (rty_1 \lessdot a), (rty_2 \lessdot a) \,\}) \\
\qquad \textit{where } a \textit{ is a fresh type variable}
\end{array}
$$

For the statements Assign, New and MethodCall the function **TYPEStmt** is given as:

$$
\begin{array}{l}
\textbf{TYPEStmt}(\,Ass, stmt\,) = \\
\quad \textbf{let}\ (stmt : rty, ConS) = \textbf{TYPEExpr}(\,Ass, stmt\,) \\
\quad \textbf{in}\ (stmt : \mathtt{Void}, ConS)
\end{array}
$$

Example 3.5. *We consider again the class* `Matrix` *from Example 2.12. Now we consider only the untyped function* `op`.

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

In **TYPE** *the function* **TYPEExpr** *is called with the arguments*

$$
\begin{array}{l}
AssAll = \{\, \mathtt{Fun2<R,T1,T2>.apply} : (\mathtt{T1}, \mathtt{T2}) \to \mathtt{R}, \\
\qquad\qquad \mathtt{this.op} : a_{\mathtt{op}}, \mathtt{this} : \mathtt{Matrix} \,\} \textit{ and} \\
lexpr_1 = \mathsf{Lam}(\,\mathtt{m}, \mathsf{Lam}(\,\mathtt{f}, \mathsf{MCall}(\,\mathsf{LoVar}(\,\mathtt{f}\,), \\
\qquad\qquad\qquad \mathtt{apply}(\,\mathtt{this}, \mathsf{LoVar}(\,\mathtt{m}\,)\,)\,)\,)\,).
\end{array}
$$

The result contains:

$$
\begin{array}{l}
lexpr_{1_t} = \\
\quad \mathsf{Lam}(\,\mathtt{m} : a_m, \\
\qquad \mathsf{Lam}(\,\mathtt{f} : a_f, \\
\qquad\quad \mathsf{MCall}(\,\mathsf{LoVar}(\,\mathtt{f}\,) : a_f, \\
\qquad\qquad \mathtt{apply}(\,\mathtt{this} : \mathtt{Matrix}, \\
\qquad\qquad\quad \mathsf{LoVar}(\,\mathtt{m}\,) : a_m\,)\,) : a_{app}\,) : a_{\lambda f}\,) : a_{\lambda \mathtt{m}}
\end{array}
$$

and the set of constraint:

$$
\begin{array}{l}
\{\, (a_{\lambda \mathtt{m}} \lessdot a_{\mathtt{op}}), (\mathtt{Fun1<}a_{\lambda f}, a_m\mathtt{>} \lessdot a_{\lambda \mathtt{m}}), \\
\;(\mathtt{Fun1<}a_{app}, a_f\mathtt{>} \lessdot a_{\lambda f}), (a_f \lessdot \mathtt{Fun2<}a_3, a_1, a_2\mathtt{>}), \\
\;(\mathtt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{app} \,\},
\end{array}
$$

where the indices of the type variables are named by its subterms.

We give another example to show overloading respectively overriding.

Example 3.6. *Let us consider the class*

```
class Main {
  r;
  app_m = r.m(1);
}
```

Let the set of type assumptions Ass_m *be given as in Example 3.4. With* $\mathtt{this.r} : a_r$ *and* $\mathtt{this.app\_m} : a_{\mathtt{app\_m}}$ *the constraints of the result of* **TYPE** *are given as:*

$$
\begin{array}{l}
\quad \{\, (\mathtt{Integer} \lessdot a_1), a_{\mathtt{r.m(1)}} \lessdot a_{\mathtt{app\_m}} \,\} \\
\quad \cup \{\, \textbf{overloading}(\,m, Ass_m, (a_r, (a_1), a_{\mathtt{r.m(1)}})\,)\,\} \\
= \{\, (\mathtt{Integer} \lessdot a_1), a_{\mathtt{r.m(1)}} \lessdot a_{\mathtt{app\_m}}, \\
\qquad \{\, a_r \lessdot \mathtt{A}, a_1 \lessdot \mathtt{String}, \mathtt{String} \lessdot a_{\mathtt{r.m(1)}} \,\} \\
\qquad\quad \| \, \{\, a_r \lessdot \mathtt{A}, a_1 \lessdot \mathtt{Object}, \mathtt{Object} \lessdot a_{\mathtt{r.m(1)}} \,\} \\
\qquad \vee \{\, a_r \lessdot \mathtt{A}, a_1 \lessdot \mathtt{Integer}, \mathtt{Integer} \lessdot a_{\mathtt{r.m(1)}} \,\} \\
\qquad\quad \| \, \{\, a_r \lessdot \mathtt{A}, a_1 \lessdot \mathtt{Object}, \mathtt{Object} \lessdot a_{\mathtt{r.m(1)}} \,\} \\
\qquad \vee \{\, a_r \lessdot \mathtt{A'}, a_1 \lessdot \mathtt{Object}, \mathtt{Object} \lessdot a_{\mathtt{r.m(1)}} \,\} \\
\qquad \vee \{\, a_r \lessdot \mathtt{B}, a_1 \lessdot \mathtt{Float}, \mathtt{Float} \lessdot a_{\mathtt{r.m(1)}} \,\} \,\}
\end{array}
$$

3.2.2 The function SOLVE

The function **SOLVE** determines the solutions of the set of constraints.

$\textbf{SOLVE}: \mathtt{ConstraintsSet} \to \mathtt{Constraints\_SubstSet} \times \{\, \textbf{ok}, \textbf{?} \,\}$

$$
\begin{array}{l}
\textbf{SOLVE}(\,CS\,) = \\
\quad \textbf{let}\ (CSSet, chk) = \textbf{FlatOverl}(\,CS\,) \\
\quad \textbf{in}\ (\textbf{SOLVE1}(\,CSSet\,), chk)
\end{array}
$$

The result of **SOLVE** is a pair, where **SOLVE1** determines the set of solutions of the given constraint set and chk shows if the result is safe (**ok**) or the result is unsafe (**?**) and must be checked again in the algorithm **TI** (Section 3.2.3).
The function **FlatOverl** erases the disjunctions in the set of constraints by constructing a cartesian product of the possible constraints.

$$
\begin{array}{l}
\textbf{FlatOverl}(\,CS \cup \{\, \theta \lessdot \theta' \,\}\,) = \\
\quad \textbf{let}\ (CSs, chk) = \textbf{FlatOverl}(\,CS\,) \\
\quad \textbf{in}\ (CSs \times \{\, \theta \lessdot \theta' \,\}, chk) \\
\textbf{FlatOverl}(\,CS \cup \{\, (\bigvee_{\tau \in Ty} (\|_{ty \in FTy}\, CS_{(\tau, ty)}))\,\}\,) = \\
\quad \textbf{let}\ (CSs, chk) = \textbf{FlatOverl}(\,CS\,) \\
\quad \textbf{in}\ (CSs \times \bigcup_{\tau \in Ty, ty \in FTy} CS_{(\tau, ty)}, \textbf{?}) \\
\textbf{FlatOverl}(\,CS \cup \{\, (\bigvee_{\tau \in Ty} CS_\tau)\,\}\,) = \\
\quad \textbf{let}\ (CSs, chk) = \textbf{FlatOverl}(\,CS\,) \\
\quad \textbf{in}\ (CSs \times \displaystyle\bigcup_{\tau \in Ty} CS_\tau, (chk \wedge \textbf{ok})) \\
\textbf{FlatOverl}(\,\emptyset\,) = (\emptyset, \textbf{ok})
\end{array}
$$

If in a set of constraints no overriding $\|$ is given, the solution is safe (**ok**). Otherwise the solution is unsafe (**?**), as in the function **overloading** all possible overridden types are assumed as alternatives. Furthermore it holds $\textbf{ok} \wedge \textbf{ok} = \textbf{ok}$ and otherwise $x \wedge y = \textbf{?}$.

In **SOLVE1** the type unification from [14] is called. There are two cases of results of the type unification. Either the results are in solved form, which means that all instances of the remaining type variables are correct solutions. Otherwise besides the solutions there are remaining constraints of the form $a\,R\,a'$, where a and a' are type variables. In this case all instances of type variables are correct, if they fulfill these constraints.

$$
\begin{array}{l}
\textbf{SOLVE1}(\,CSSet\,) = \\
\quad \textbf{foreach}\ cs \in CSSet \\
\qquad \textbf{let}\ subs_{cs} = \textbf{TUnify}(\,cs\,) \\
\qquad\quad \textbf{if}\ \text{(there are } \sigma \in subs_{cs} \text{ in solved form)}\ \textbf{then} \\
\qquad\qquad \bigcup_{cs} \{\, c \in subs_{cs} \mid c \text{ is in solved form} \,\} \\
\qquad\quad \textbf{if}\ \text{(there are } \sigma \in subs_{cs}\text{, which has the form} \\
\qquad\qquad \{\, v\,R\,v' \mid v, v' \text{are type vars} \,\} \\
\qquad\qquad\quad \cup \{\, v \doteq \theta \mid v \text{ is a type vars} \,\})\ \textbf{then} \\
\qquad\qquad \bigcup_{cs} \{\, c \in subs_{cs} \mid c \text{ has the given form} \,\}
\end{array}
$$

Finally both functions **TYPE** and **SOLVE** are combined to the type inference algorithm by the function **TI**.

3.2.3 The type inference algorithm TI

The type inference algorithm for Java 8 **TI** calls first the function **TYPE**. The function **TYPE** inserts type annotations, widely type variables as placeholders, in the Java class and determines a set of type constraints. Second the function **SOLVE** solves the type constraints by type unification. Finally the set of substitutions, which are results of **SOLVE**, are applied to the type annotated Java class. The result of **TI** is a set of pairs of a remaining set of constraints and a typed Java 8 class.

$$\mathbf{TI}\colon \texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \{\, (\texttt{Constraints}, \texttt{TClass}) \,\}$$

$$
\begin{array}{l}
\mathbf{TI}(Ass, \mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}, \mathit{mdecls}\,)\,) = \\
\quad \mathbf{let} \\
\qquad (\mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}_t, \mathit{mdecls}_t\,), ConS) = \\
\qquad\quad \mathbf{TYPE}(Ass, \mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}, \mathit{mdecls}\,)\,) \\
\qquad (\{\,(cs_1, \sigma_1), \ldots, (cs_n, \sigma_n)\,\}, chk) = \mathbf{SOLVE}(\,ConS\,) \\
\quad \mathbf{in} \\
\qquad \{\,(cs_i, \sigma_i(\,\mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}_t, \mathit{mdecls}_t\,)\,))\\
\qquad\qquad\qquad\qquad\qquad\qquad \mid 1 \leqslant i \leqslant n \,\}
\end{array}
$$

The result of **TI** is a set of typed Java 8 classes with constraints. As Java does not contain type constraints, we consider as results all programs, where the instances of the type variables fulfill the constraints. For overridden methods ($chk = \mathbf{?}$) the types of the instances must be checked by the **MethodCall**–rule, as in the function **overloading** all possible overridden types are assumed as alternatives.

Example 3.7. *We continue Example 3.5. The set of constraints is given as:*

$$
\begin{array}{l}
CSSet = \{\, \{\, (a_{\lambda\mathtt{m}} \lessdot a_{\mathtt{op}}), (\texttt{Fun1<}a_{\lambda f}, a_m\texttt{>} \lessdot a_{\lambda\mathtt{m}}), \\
\qquad (\texttt{Fun1<}a_{app}, a_f\texttt{>} \lessdot a_{\lambda f}), (a_f \lessdot \texttt{Fun2<}a_3, a_1, a_2\texttt{>}), \\
\qquad (\texttt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{app}) \,\}\,\}
\end{array}
$$

For cs $\in CSSet$ *call of* **TUnify:** *With step 4 (***TUnify***) we get:*

$$
\begin{array}{l}
\{\{a_{\lambda\mathtt{m}} \lessdot a_{\mathtt{op}}, a_{\lambda\mathtt{m}} \doteq \texttt{Fun1<}a_{\lambda f}, a_m\texttt{>}, \\
\quad a_{\lambda f} \doteq \texttt{Fun1<}a_{app}, a_f\texttt{>}, a_f \doteq \texttt{Fun2<}a_3, a_1, a_2\texttt{>}, \\
\quad a_1 \doteq \texttt{Matrix}, a_m \lessdot a_2, a_3 \lessdot a_{app} \,\}, \\
\ \{a_{\lambda\mathtt{m}} \lessdot a_{\mathtt{op}}, a_{\lambda\mathtt{m}} \doteq \texttt{Fun1<}a_{\lambda f}, a_m\texttt{>}, \\
\quad a_{\lambda f} \doteq \texttt{Fun1<}a_{app}, a_f\texttt{>}, a_f \doteq \texttt{Fun2<}a_3, a_1, a_2\texttt{>}, \\
\quad a_1 \doteq \texttt{Vec<Vec<Int>>}, a_m \lessdot a_2, a_3 \lessdot a_{app} \,\}\,\}
\end{array}
$$

With step 5 (subst) and step 6 of **TUnify** *we get:*

$$
\begin{array}{l}
\{(\{a_m \lessdot a_2, a_3 \lessdot a_{app}\,\}, \\
\quad \{a_{\mathtt{op}} \doteq \texttt{Fun1<Fun1<a}_{app}\texttt{, Fun2<a}_3\texttt{, Matrix, a}_2\texttt{>>, a}_\mathtt{m}\texttt{>}, \\
\quad a_{\lambda\mathtt{m}} \doteq \texttt{Fun1<Fun1<a}_{app}\texttt{, Fun2<a}_3\texttt{, Matrix, a}_2\texttt{>>, a}_\mathtt{m}\texttt{>}, \\
\quad a_{\lambda f} \doteq \texttt{Fun1<a}_{app}\texttt{, Fun2<a}_3\texttt{, Matrix, a}_2\texttt{>>}, \\
\quad a_f \doteq \texttt{Fun2<a}_3\texttt{, Matrix, a}_2\texttt{>}, a_1 \doteq \texttt{Matrix}\,\}) \\
\ (\{a_m \lessdot a_2, a_3 \lessdot a_{app}\,\}, \\
\quad \{a_{\mathtt{op}} \doteq \texttt{Fun1<Fun1<a}_{app}\texttt{, Fun2<a}_3\texttt{, Vec<Vec<Int>>, a}_2\texttt{>>, a}_\mathtt{m}\texttt{>}, \\
\quad a_{\lambda\mathtt{m}} \doteq \texttt{Fun1<Fun1<a}_{app}\texttt{, Fun2<a}_3\texttt{, Vec<Vec<Int>>, a}_2\texttt{>>, a}_\mathtt{m}\texttt{>}, \\
\quad a_{\lambda f} \doteq \texttt{Fun1<a}_{app}\texttt{, Fun2<a}_3\texttt{, Vec<Vec<Int>>, a}_2\texttt{>>}, \\
\quad a_f \doteq \texttt{Fun2<a}_3\texttt{, Vec<Vec<Int>>, a}_2\texttt{>}, a_1 \doteq \texttt{Vec<Vec<Int>>}\,\})\,\}
\end{array}
$$

One result is given as the instances $\{\, a_m \mapsto \texttt{Matrix}, a_2 \mapsto \texttt{Matrix}, a_3 \mapsto \texttt{Matrix}, a_{app} \mapsto \texttt{Matrix} \,\}$. *It is obvious that the constraints are fulfilled. As there is no overriding (*$chk = \mathbf{ok}$*) this is a result. This result corresponds to the typing in Example 2.12.*

But there is another result which is more general.

```
class Matrix<T2, T1 extends T2, T4, T3 extends T4>
extends Vector<Vector<Integer>> {
  Fun1<Fun1<T4, Fun2<T3, Matrix,T2>>, T1>
  op = (T1 m) -> (Fun2<T3, Matrix,T2> f) ->
                                  f.apply(this, m); }
```

The corresponding instances are given as $\{\, a_m \mapsto \texttt{T1}, a_2 \mapsto \texttt{T2}, a_3 \mapsto \texttt{T3}, a_{app} \mapsto \texttt{T4} \,\}$. *The instances fulfill the constraints.*

Example 3.8. *We continue Example 3.6. First the function* ***FlatOverl*** *has to be applied, which leads to*

$$
\begin{array}{ll}
CSSet = \{\, \{\, (\texttt{Integer} \lessdot a_1), (a_{\mathtt{r.m(1)}} \lessdot a_{\mathtt{app\_m}}), & \\
\qquad (a_r \lessdot \texttt{A}), (a_1 \lessdot \texttt{String}), (\texttt{String} \lessdot a_{\mathtt{r.m(1)}})\,\}, & (1) \\
\quad \{(\texttt{Integer} \lessdot a_1), (a_{\mathtt{r.m(1)}} \lessdot a_{\mathtt{app\_m}}), & \\
\qquad (a_r \lessdot \texttt{A}), (a_1 \lessdot \texttt{Object}), (\texttt{Object} \lessdot a_{\mathtt{r.m(1)}})\,\}, & (2) \\
\quad \{(\texttt{Integer} \lessdot a_1), (a_{\mathtt{r.m(1)}} \lessdot a_{\mathtt{app\_m}}), & \\
\qquad (a_r \lessdot \texttt{A}), (a_1 \lessdot \texttt{Integer}), (\texttt{Integer} \lessdot a_{\mathtt{r.m(1)}})\,\}, & (3) \\
\quad \{(\texttt{Integer} \lessdot a_1), (a_{\mathtt{r.m(1)}} \lessdot a_{\mathtt{app\_m}}), & \\
\qquad (a_r \lessdot \texttt{A}'), (a_1 \lessdot \texttt{Object}), (\texttt{Object} \lessdot a_{\mathtt{r.m(1)}})\,\}, & (4) \\
\quad \{(\texttt{Integer} \lessdot a_1), (a_{\mathtt{r.m(1)}} \lessdot a_{\mathtt{app\_m}}), & \\
\qquad (a_r \lessdot \texttt{B}), (a_1 \lessdot \texttt{Float}), (\texttt{Float} \lessdot a_{\mathtt{r.m(1)}})\,\}\,\} & (5)
\end{array}
$$

$chk = \mathbf{?}$

In **SOLVE1** *the type unification* **TUnify** *is applied to each element* $cs \in CSSet$. *The result of* **SOLVE1** *is:*

$$
\begin{array}{ll}
\{\ \{\,(a_{\mathtt{r}} \doteq \texttt{A}), (a_1 \doteq \texttt{Integer}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), & \\
\qquad (a_{\mathtt{r.m(1)}} \doteq \texttt{Object})\,\}, & (\mathit{from}(2)) \\
\quad \{\,(a_{\mathtt{r}} \doteq \texttt{A}), (a_1 \doteq \texttt{Object}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), & \\
\qquad (a_{\mathtt{r.m(1)}} \doteq \texttt{Object})\,\}, & (\mathit{from}(2)) \\
\quad \{\,(a_{\mathtt{r}} \doteq \texttt{A}), (a_1 \doteq \texttt{Integer}), (a_{\mathtt{app\_m}} \doteq \texttt{Integer}), & \\
\qquad (a_{\mathtt{r.m(1)}} \doteq \texttt{Integer})\,\}, & (\mathit{from}(3)) \\
\quad \{\,(a_{\mathtt{r}} \doteq \texttt{A}), (a_1 \doteq \texttt{Integer}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), & \\
\qquad (a_{\mathtt{r.m(1)}} \doteq \texttt{Integer})\,\}, & (\mathit{from}(3)) \\
\quad \{\,(a_{\mathtt{r}} \doteq \texttt{A}'), (a_1 \doteq \texttt{Object}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), & \\
\qquad (a_{\mathtt{r.m(1)}} \doteq \texttt{Object})\,\}, & (\mathit{from}(4)) \\
\quad \{\,(a_{\mathtt{r}} \doteq \texttt{A}'), (a_1 \doteq \texttt{Integer}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), & \\
\qquad (a_{\mathtt{r.m(1)}} \doteq \texttt{Object})\,\} & (\mathit{from}(4)) \\
\ \} &
\end{array}
$$

As there are overridings ($chk = \mathbf{?}$*) the results must be checked by the* **MethodCall***–rule. The correct results are then given as:*

$$
\begin{array}{l}
\{\{(a_{\mathtt{r}} \doteq \texttt{A}), (a_1 \doteq \texttt{Integer}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), (a_{\mathtt{r.m(1)}} \doteq \texttt{Object})\}, \\
\ \{(a_{\mathtt{r}} \doteq \texttt{A}), (a_1 \doteq \texttt{Integer}), (a_{\mathtt{app\_m}} \doteq \texttt{Integer}), \\
\ (a_{\mathtt{r.m(1)}} \doteq \texttt{Integer})\}, \\
\ \{(a_{\mathtt{r}} \doteq \texttt{A}), (a_1 \doteq \texttt{Integer}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), (a_{\mathtt{r.m(1)}} \doteq \texttt{Integer})\}, \\
\ \{(a_{\mathtt{r}} \doteq \texttt{A}'), (a_1 \doteq \texttt{Object}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), (a_{\mathtt{r.m(1)}} \doteq \texttt{Object})\} \\
\ \{(a_{\mathtt{r}} \doteq \texttt{A}'), (a_1 \doteq \texttt{Integer}), (a_{\mathtt{app\_m}} \doteq \texttt{Object}), (a_{\mathtt{r.m(1)}} \doteq \texttt{Object})\} \\
\}
\end{array}
$$

Theorem 3.9 (Correctness and Completeness). *Let a set of type assumptions* Ass *and a class* τ *be given. Then the following is equivalent:*

- $\mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}_r, \mathit{mdecls}_r\,))$ *with*
  $\mathit{fdecls}_r =$
  $$[\mathsf{Field}(\,t_1, f_1, \mathit{lexpr}_{1_r}\,), \ldots, \mathsf{Field}(\,t_n, f_n, \mathit{lexpr}_{n_r}\,)]$$
  $\mathit{mdecls}_r =$
  $$
  \begin{array}{l}
  [\mathsf{Method}(\,t'_1, \mathit{me}_1, (v_1 : t'_{11}), \ldots, (v_{m_1} : t'_{1m_1}), \mathit{bl}_{1_r}\,) \\
  , \ldots, \\
  \mathsf{Method}(\,t'_m, \mathit{me}_m, (v_1 : t'_{m1}), \ldots, (v_{m_m} : t'_{mm_m}), \mathit{bl}_{m_r}\,)]
  \end{array}
  $$
  is a result of the type inference algorithm $\mathbf{TI}(\,Ass, \tau\,)$, *where the type variables are instanced such they fulfill the constraints.*
- *There are equivalent types* $\bar{t}_i \sim_{\mathit{fi}} t_i$, $\bar{t}'_{jk} \sim_{\mathit{fi}} t'_{jk}$ *and* $\bar{t}'_j \sim_{\mathit{fi}} t'_j$ *such that with* $O_\theta = \{\, id : ty \mid \theta.id : ty \in Ass \,\}$ *for* $\theta \neq \tau$ *and* $O_\tau = \{\, f_i : \bar{t}_i \mid 1 \leqslant i \leqslant n \,\} \cup \{\, m_j : (\bar{t}'_{j1}, \ldots, \bar{t}'_{jm_j}) \rightarrow \bar{t}'_j \mid 1 \leqslant j \leqslant m \,\}$ *holds for* $1 \leqslant i \leqslant n : (O, \tau, \tau') \rhd_{Expr}\ \mathit{lexpr}_i : \bar{t}_i$ *respectively* $(O, \tau, \tau') \rhd_{Stmt}\ \mathit{lexpr}_i : \bar{t}_i$ *and with* $O_\tau = \{\, f_i : \bar{t}_i \mid 1 \leqslant i \leqslant n \,\} \cup \{\, m_j : (\bar{t}'_{j1}, \ldots, \bar{t}'_{jm_j}) \rightarrow \bar{t}'_j \mid 1 \leqslant j \leqslant m \,\} \cup \{\, v_k : \bar{t}'_{jk} \mid 1 \leqslant j \leqslant m, 1 \leqslant k \leqslant m_j \,\}$ *holds for* $1 \leqslant j \leqslant m$: $(O, \tau, \tau') \rhd_{Stmt}\ \mathit{bl}_j : \bar{t}'_j$.

Proof. We will give a sketch of the prove.

1. Step: By induction we prove that the solution of the resulting constraints set of **TYPE** is equivalent to the derivation of the types of the corresponding expressions respectively statements by $\triangleright_{Expr}$ respectively $\triangleright_{Stmt}$.

2. Step: The algorithm **TUnify** is correct and complete and solve the constraints set.

Two exceptions must be considered. On the one hand all overridden methods are assumed as possible method calls (function **overloading**). Therefore some derived types from the results of **TI** are not type correct. These must be erased, which is done by checking with the **MethodCall**–rule.
On the other hand results of the algorithm **TUnify**, which contains pairs $v\,R\,v'$, where v and v' are type variables, are considered in **SOLVE1**. These pairs are not considered in [14]. We prove that instances, which solve the constraints are type unifiers. Furthermore we prove, that for all other results of **TUnify**, which are not in solved form (all pairs of the form $v \doteq \theta$), there is no type unifier. □

4. Related Work

The programming language Scala [11] allows functional programming features similar to Java$_\lambda$. In Scala functions are also first-class citizens. It supports lambda expressions as well as higher-order functions. In addition to Java Scala allows real function types as in Java$_\lambda$, currying and pattern-matching.
In comparison to our approach, Scala contains indeed a type–inference system. But the type–inference system is restricted to *local type inference* [12], which means that often type declarations of variables and result types of methods can be omitted. For complete lambda expressions and recursive methods, it is not possible to infer the result types.
In C# (e.g. [17]) closures are also included. Function types are given as *delegates*. Delegates are similar to function pointers in C or C++. A delegate defines a type that encapsulates a method with argument types and a return type. A delegate plays the role in C# as a functional interfaces in Java 8. In C# there is no type inference.
The basis of all type inference systems is the approach of Hindley, Damas and Milner [4, 9, 10]. They described a type inference system for a lambda calculus with parameteric polymorphic types, but without subtyping. There are many extensions of this Hindley-Milner approach. One extension is the approach of Fuh and Mishra [5], which we used in the type inference algorithm for Java$_\lambda$ [15]. Another extension is the approach of Aiken and Wimmer [3]. They consider type inclusion constraints, which are comparable to our type constraints, but their type system contains additionally intersection types, union types and function types.

5. Conclusion and future work

We have considered the Java 8 extensions closures and functional interfaces. We defined first an equivalence relation on functional interfaces and gave type inference rules for a core of Java 8. Finally we presented a type inference algorithm for Java 8, which infers types for complete closures and methods. The algorithm is an enhancement of the approaches of Fuh and Mishra [5] and of our approaches [13, 14].
In the future, we will give a principal typing property [18] for Java 8. That is why the type system must by extended, such that type constraints on type variables are introduced. The constraints can be introduced in Java as class parameters, similar as in Example 3.7. Therefore it must be allowed to give a type variable multiple times in the parameter list. Additionally, a class parameter inference can be introduced, where the class parameters are given as the remaining type variables of the result of the type inference algorithm. Finally an IDE has to be developed, which supports the user by automatic type inference, similar as we have done for Java 5.0 [13].
Another approach could be the enhancement of the type inference for Java$_\lambda$. The main difference between Java$_\lambda$ and Java 8 is that Java$_\lambda$ has real function types. The idea is to use our type unification [14] in the type inference algorithm for Java$_\lambda$ [15]. It would improve the results, such that they are not any longer unsignificant well-typings, but unique types with some type constraints.

References

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

[2] Lambda specification. 2011. URL `http://jcp.org/aboutJava/communityprocess/edr/jsr335/index.html`. Version 0.4.2.

[3] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.

[5] Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Proceedings 2nd European Symposium on Programming (ESOP '88)*, pages 94–114, 1988.

[6] B. Goetz. State of the lambda. 10 October 2010. URL `http://cr.openjdk.java.net/~Briangoetz/lambda/lambda-state-3.html`.

[7] B. Goetz. State of the lambda. December 2011. URL `http://cr.openjdk.java.net/~Briangoetz/lambda/lambda-state-4.html`.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.

[9] R. Hindley. The principle type scheme of an object in combinatory logic. *Trans. Am. Math. Soc. 146*, pages 29–60, December 1969.

[10] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–378, 1978.

[11] M. Odersky. *The Scala language specication version 2.9*. http://www.scala-lang.org/docu/files/ScalaReference.pdf, May 2011. Draft.

[12] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. *POPL 2001 Proc. 28th ACM Symposium on Principles of Programming Languages*, 36(3):41–53, 2001.

[13] M. Plümicke. Typeless Programming in Java 5.0 with wildcards. In V. Amaral, L. Veiga, L. Marcelino, and H. C. 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.

[14] M. Plümicke. Java type unification with wildcards. In D. Seipel, M. Hanus, and A. Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

[15] M. 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. ISBN 978-1-4503-0935-6. doi: http://doi.acm.org/10.1145/2093157.2093171.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, Jan. 1965.

[17] J. Skeet. *C# in Depth*. Manning Publications Co., second edition, 2010.

[18] S. van Bakel. Principal type schemes for the strict type assignment system. *Journal of Logic and Computing*, 3(6):643–670, 1993.