Well-typings for Java_λ
– The algorithm –

Martin Plümicke

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

Abstract. In the last decade Java has been extended by some features, which are well-known from functional programming languages. In Java 8 the language will be expanded by closures (λ–expressions).
In our contribution we give a formal definition for an abstract syntax of a reduced language Java_λ with closures, define the type system, and formalize the subtyping relation.
We define the set of types as an extension of the generic type definition for Java 5 types.
Finally we present a type inference algorithm. The type inference algorithm is an adaptation of a type inference algorithm for a typed λ–calculus.
The inferred types are well-typings. A well-typing is a conditional type for an expression, where the conditions are given by a set of consistent coercions.

1 Introduction

In several steps the Java type system is extended by features, which we know from functional programming languages. In Java 5.0 [GJSB05] generic types are introduced. Furthermore, a reduced form of existential types (bounded wildcards) is introduced. For Java 8 it is announced, that closures (λ–expressions) should be introduced. In October 2010 [Goe10] it was planned, against previous announcements, to omit function types. The reason was that function types could confuse the programmer in several ways.
Instead SAM types should be used. A SAM type is either an interface which declares just one method or an abstract class which declares just one abstract method.
Let us consider an example: The interface `Operation` is given by

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

To call the method op in Java typically an anonymous inner class is created, e.g.:

```
foo.doAddition(new Operation () {
                   public int op (int x, int y) {
                       return x + y;
                   }
               });
```

λ–expressions could simplify the call by using

```
foo.doAddition(#{ (int x, int y) -> x + y })
```

In earlier announcements (e.g. [lam10]) explicite function types had been introduced, which would mean, that

```
#int(int, int)
```

could be used instead of the interface declaration `Operation`.
If we consider the SAM declaration

```
void doAddition(Operation o) { ... }
```

in comparison to the function type declaration

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

the SAM declaration is obviously more readable. But an additional declaration of the interface `Operation` is necessary.
This means, that it would be more convienent for writing methods with function types as arguments to use the function types, explicitly. Especially the use of higher order functions would cause the declaration of an interface `Func`.

```
interface Func<Y,X> { public Y f (X x) }
```

Let us consider the type `##int(#int(int,int))(int)`. For the type `#int(int,int)` an interface

```
interface A { public  int g (int x, int y) }
```

is needed. The correspondig type in SAM representation is

```
Func<Func<Integer, A>, Integer>
```

In functional programming languages the same problem arises, which is solved by the mechanism of type inference. This means that no explicite type annotations are given and the compiler infers the type automatically. In our example the declaration would be given as

```
doAddition(o) { ... }
```

and the compiler would infer the type

$$\texttt{doAddition} : \#\texttt{void}(\#\texttt{int}(\texttt{int}, \texttt{int})).$$

This approach allows function types and readable code at the same time.
This is the reason why we developed a Java type inference system which assists the programmer by calculating types automatically. This type inference system allows us to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principal types. In [Plü07] we presented a type inference algorithm for a core Java 5.0 language.
The extension of function types in Java is considered in several contributions. Our approach is following the Java language specification [lam10]. This approach is limited following Mark Reinhold's Blog (Principal Engineer Java SE and OpenJDK) to two key features:

- A literal syntax, for writing closures, and
- Function types, so that closures are first-class citizens in the type system.

Additionally, to integrate closures with the rest of the language and the platform two more features are needed:

- Closure conversion to implement a single-method interface or abstract class and
- Extension methods.

Our goal is to extend the type inference algorithm to the features of Java 8. For this we consider the type inference algorithm of Fuh and Mishra [FM88]. This algorithm infers types for a type system of a λ–calculus with subtyping and without additional overloading. The Java 8 preconditions are very similar.
We will recognize, that the preconditions of Java 8 in general lead to principal typings, which are called <u>well-typings</u> by Fuh and Mishra. A well-typing is a conditional type for an expression, where the conditions are given by a set of consistent coercions (constraints).
The paper is organized as follows. In the next section we define an abstract syntax for a reduced language Java$_\lambda$. In the third section we consider the Java$_\lambda$ types and the subtyping relation. In the forth section we consider the adaptation of the Fuh and Mishra's type inference algorithm. Finally we close with a summary and an outlook.

2 The language

The language Java$_\lambda$ (Fig. 1) is an abstract representation of a core of Java 8. It is an extension of our language in [Plü07]. The additional features are underlined. Beside instance variables functions can be declared in classes. A function

$$
\begin{array}{lll}
\mathit{Source} & := & \mathit{class}* \\
\mathit{class} & := & \mathsf{Class}(\mathit{simpletype}, [\ \mathsf{extends}(\ \mathit{simpletype}\ ),]\ \mathit{IVarDecl}*, \underline{\mathit{FunDecl}*}) \\
\mathit{IVarDecl} & := & \mathsf{InstVarDecl}(\ \mathit{simpletype}, \mathit{var}\ ) \\
\underline{\mathit{FunDecl}} & := & \underline{\mathsf{Fun}(\ \mathit{fname}, [\boldsymbol{type}], \mathit{lambdaexpr}\ )} \\
\mathit{block} & := & \mathsf{Block}(\ \mathit{stmt}*\ ) \\
\mathit{stmt} & := & \mathit{block}\ \mid\ \mathsf{Return}(\ \mathit{expr}\ )\ \mid\ \mathsf{While}(\ \mathit{bexpr}, \mathit{block}\ ) \\
 & \mid & \mathsf{LocalVarDecl}(\ \mathit{var}[, \boldsymbol{type}]\ )\ \mid \mathsf{If}(\ \mathit{bexpr}, \mathit{block}[, \mathit{block}]\ )\ \mid\ \mathit{stmtexpr} \\
\underline{\mathit{lambdaexpr}} & := & \underline{\mathsf{Lambda}(\ ((\mathit{var}[, \boldsymbol{type}]))*, (\mathit{stmt}\ \mid\ \mathit{expr})\ )} \\
\mathit{stmtexpr} & := & \mathsf{Assign}(\ \mathit{vexpr}, \mathit{expr}\ )\ \mid\ \mathsf{New}(\ \mathit{simpletype}, \mathit{expr}*\ )\ \mid\ \underline{\mathsf{Eval}(\ \mathit{expr}, \mathit{expr}*\ )} \\
\mathit{vexpr} & := & \mathsf{LocalOrFieldVar}(\ \mathit{var}\ )\ \mid\ \mathsf{InstVar}(\ \mathit{expr}, \mathit{var}\ ) \\
\mathit{expr} & := & \underline{\mathit{lambdaexpr}}\ \mid\ \mathit{stmtexpr}\ \mid\ \mathit{vexp}\ \mid\ \mathsf{this}\ \mid\ \mathsf{This}(\ \mathit{simpletype}\ )\ \mid\ \mathsf{super} \\
 & \mid & \underline{\mathsf{InstFun}(\ \mathit{expr}, \mathit{fname}\ )}\ \mid\ \mathit{bexp}\ \mid\ \mathit{sexp}
\end{array}
$$

Fig. 1. The abstract syntax of Java_λ

is declared by its name, optionally its type, and the λ–expression. Methods are not considered in this framework, as methods can be expressed by functions. A λ–expression consists of an optionally typed variable and either an statement or an expression. Furthermore, the statement expressions respectively the expressions are extended by evaluation-expressions, the λ–expressions, and instances of functions.

The concrete syntax in this paper of the λ–expressions is oriented at [Goe10], while the concrete syntax of the function types and closure evaluation is oriented at [lam10].

The optional type annotations [***type***] are the types, which can be inferred by the type inference algorithm.

3 Types and subtyping

As a base for the type inference algorithm we have to make a formal definition of the Java 8 types. First we give again the definition of simple types (first-order types). The definition is connected to the corresponding definitions in [GJSB05], Section 4.5. and [Plü07], Section 2.

Definition 1 (Simple types). *Let $BTV^{(ty)}$ be the set of bounded type variables and TC a $(BTV)^*$–indexed set of type constructors (class names). Then, the set of simple types $\mathsf{SType}_{TS}(\ BTV\ )$ for the given type signature $(\mathsf{SType}_{TS}(\ BTV\ ),\ TC)$ is defined as the smallest set satisfying the following conditions:*

- *For each type ty:* $\underline{BTV^{(ty)}} \subseteq \mathsf{SType}_{TS}(\ BTV\ )$
- $\underline{TC^{()}} \subseteq \mathsf{SType}_{TS}(\ BTV\ )$

– *For* $ty_i \in \mathsf{SType}_{TS}(\,BTV\,)$
$\cup\{\,\texttt{?}\,\}$
$\cup\{\,\texttt{? extends}\ \tau \mid \tau \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$
$\cup\{\,\texttt{? super}\ \tau \mid \tau \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$
and $C \in TC^{(a_1|_{b_1} \ldots a_n|_{b_n})}$ *holds*

$$\underline{C\texttt{<}ty_1, \ldots, ty_n\texttt{>}} \in \mathsf{SType}_{TS}(\,BTV\,)$$

if after $C\texttt{<}ty_1, \ldots, ty_n\texttt{>}$ *subjected to the capture conversion resulting in the type* $C\texttt{<}\overline{ty_1}, \ldots, \overline{ty_n}\texttt{>}$[1], *for each actual type argument* $\overline{ty_i}$ *holds:*

$$\overline{ty_i} \leq^* b_i[a_j \mapsto \overline{ty_j} \mid 1 \leqslant j \leqslant n],$$

where $\leq^*$ *is a subtyping ordering (Def. 3).*

– *The* *<u>set of implicit type variables with lower or upper bounds</u>* *belongs to* $\mathsf{SType}_{TS}(\,BTV\,)$

Simple types are the first-order base-types of the Java_λ type system. Simple types are used in Java_λ explictly in the extension relation, which defines the subtyping ordering.
The set of simple types is extended to the set of Java_λ types by adding function types.

Definition 2 (Types). *Let* $\mathsf{SType}_{TS}(\,BTV\,)$ *be a set of simpletypes. The set of* Java_λ *types* $\mathsf{Type}_{TS}(\,BTV\,)$ *is defined by*

– $\mathsf{SType}_{TS}(\,BTV\,) \subseteq \mathsf{Type}_{TS}(\,BTV\,)$
– *For* $ty, ty_i \in \mathsf{Type}_{TS}(\,BTV\,)$

$$\#\,ty\,(ty_1, \ldots, ty_n) \in \mathsf{Type}_{TS}(\,BTV\,)^2$$

Analogously, the definition of the subtyping relation on simple types is extended to the subtyping relation on Java_λ types.

Definition 3 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(\,BTV\,)$). *Let* $TS = (\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *be a type signature of a given* Java *program and* $<$ *the corresponding extends relation. The* *<u>subtyping relation</u>* $\leq^*$ *is given as the reflexive and transitive closure of the smallest relation satisfying the following conditions:*

– *if* $\theta < \theta'$ *then* $\theta \leq^* \theta'$.
– *if* $\theta_1 \leq^* \theta_2$ *then* $\sigma_1(\,\theta_1\,) \leq^* \sigma_2(\,\theta_2\,)$ *for all substitutions* $\sigma_1,\ \sigma_2 : BTV \rightarrow \mathsf{SType}_{TS}(\,BTV\,)$*, where for each type variable* a *of* θ_2 *holds* $\sigma_1(\,a\,) = \sigma_2(\,a\,)$ *(soundness condition).*
– $a \leq^* \theta_i$ *for* $a \in BTV^{(\theta_1 \& \ldots \& \theta_n)}$ *and* $1 \leqslant i \leqslant n$

[1] For non wildcard type arguments the capture conversion $\overline{ty}_i$ equals ty_i
[2] Often function types $\#\,ty\,(ty_1, \ldots, ty_n)$ are written as $(ty_1, \ldots, ty_n) \rightarrow ty$.

– *It holds* $C\texttt{<}\theta_1,\ldots,\theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1,\ldots,\theta'_n\texttt{>}$ *if for each* θ_i *and* θ'_i*, respectively, one of the following conditions is valid*[3]
  - $\theta_i = {}_?\overline{\theta}_i$, $\theta'_i = {}_?\overline{\theta}'_i$ *and* $\overline{\theta}_i \leq^* \overline{\theta}'_i$.
  - $\theta_i = {}^?\overline{\theta}_i$, $\theta'_i = {}^?\overline{\theta}'_i$ *and* $\overline{\theta}'_i \leq^* \overline{\theta}_i$.
  - $\theta_i, \theta'_i \in \mathsf{SType}_{TS}(\,BTV\,)$ *and* $\theta_i = \theta'_i$
  - $\theta'_i = {}_?\theta_i$
  - $\theta'_i = {}^?\theta_i$

  (*cp. [GJSB05] §4.5.1.1 type argument containment*)
– *Let* $C\texttt{<}\overline{\theta}_1,\ldots,\overline{\theta}_n\texttt{>}$ *be the capture conversions of* $C\texttt{<}\theta_1,\ldots,\theta_n\texttt{>}$ *and* $C\texttt{<}\overline{\theta}_1,\ldots,\overline{\theta}_n\texttt{>} \leq^* C\texttt{<}\theta'_1,\ldots,\theta'_n\texttt{>}$ *then holds*

$$C\texttt{<}\theta_1,\ldots,\theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1,\ldots,\theta'_n\texttt{>}.$$

– $\theta \leq^* {}_{\theta|}T$

Definition 4 (Subtyping relation $\leq^*$ on $\mathsf{Type}_{TS}(\,BTV\,)$). *Let* $\leq^*$ *be a subtyping relation on simple types* $\mathsf{SType}_{TS}(\,BTV\,)$*. Then, the extension to* $\mathsf{Type}_{TS}(\,BTV\,)$ *is defined as:*

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

Example 1. Let the following Java_λ program be given.

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

  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = #{ Matrix m -> #{ #Matrix(Matrix, Matrix) f -> f(Matrix.this, m)}}

  #Matrix(Matrix, Matrix)
  mul = #{ (Matrix m1, Matrix m2) ->
    Matrix ret = new Matrix ();
      for(int i = 0; i < size(); i++) {
          Vector<? extends 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.(m2).(m1.mul);}
}
```

[3] ${}_?\theta$ means '`? extends` θ', while ${}^?\theta$ means '`? super` θ'.

`op` is a curried function with two arguments. The first one is a matrix and the second one is a function which takes two matrices and returns another matrix. The function `op` applies its second argument to its own object and its first argument.
`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`.

4 Type inference algorithm

In the late eighties Fuh and Mishra [FM88] gave an algorithm for type inference in a λ-calculus with subtyping. Their calculus indeed allows subtyping but no additional overloading. This means that the type system of Java_λ and the type system of their language are equivalent. There are only some differences in the subtyping relation: By Fuh and Mishra the subtyping relation is not defined on user defined type constructors (classes) and the relation is always finite, while in Java_λ user defined classes and infinite chains in the subtyping relation are allowed. Infinite chains are induced by wildcards. This means that the usual unification [MM82], which is used by Fuh and Mishra, cannot be used in our approach. We will replace it by our type unification algorithm of [Plü09].

In this paper we present all algorithms 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.

4.1 Summary of Fuh and Mishra's algorithm

The algorithm is called WTYPE, which stands for well-typing. A well-typing is a data-structure $C, A \vdash N : t$, where C is a set of consistent coercions (constraints), A is a set of type assumptions, N is an expression, and t is the derived type.

The algorithm is given as:

$$
\begin{array}{l}
\mathbf{WTYPE} : \texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{WellTyping} + \{\, \mathit{fail} \,\} \\
\mathbf{WTYPE}(\, A, e \,) = \mathbf{let}\ (\theta, C) = \mathbf{TYPE}(\, A, e \,)\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{let}\ \sigma = \mathbf{MATCH}(\, C \,)\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{let}\ C' = \mathbf{SIMPLIFY}(\, \sigma(\, C \,)\,)\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{if}\ \mathbf{CONSISTENT}(\, C' \,)\ \mathbf{then} \\
\qquad\qquad\qquad\qquad (C', A) \vdash e : \sigma(\, \theta \,) \\
\qquad\qquad\qquad\quad \mathbf{else} \\
\qquad\qquad\qquad\qquad \mathit{fail}
\end{array}
$$

This means that the result of the algorithm is a type with a set of constraints. **WTYPE** consists of four functions:

TYPE : `TypeAssumptions` $\times$ `Expression`$\rightarrow$ `Type` $\times$ `CoercionSet` maps a fresh type variable to each subterm of the input expression and determines coercions, which contain the function and the tuple constructors derived from the structure of the λ–expression.

MATCH : `CoercionSet` → `Substitution` + $\{ \mathit{fail} \}$ determines a substitution σ. σ applied to the input coercion set C results in a minimal matching instance of C. The algorithm **MATCH** is an adaptation to coercions of the Martelli, Montanari unification algorithm [MM82].

SIMPLIFY : `CoercionSet` → `ACoercionSet` eliminates the type constructors, especially the function and the tuple-constructor. The result is a set of atomic coercions. An atomic coercion is a coercion of two simple types, where at least one is a type variable.

CONSISTENT : `CoercionSet` → `Boolean` checks if a set of atomic coercions is consistent, by determining all possible instances of each variable. If finally for each variable there is a non-empty set of instances, the set of atomic coercions is consistent.

4.2 Adaptation to the Java$_\lambda$ type system

Now we give the adaptation of the type inference algorithm to Java$_\lambda$.

The function TYPE The function **TYPE** is applied to a set of type assumptions of known classes and a new class. The result is a set of type assumptions, where for each defined function a mapped type variable, and a set of coercions (denoted by $a \lessdot a'$) is given.
The argument set of type assumptions contains four different forms of elements:

$v : \theta$**:** Assumptions for local or instance variables of the actual class.
$f : \#\,\theta\,(\theta_1, \ldots, \theta_n)$**:** Assumptions for functions of the actual class.
$\tau.v : \theta$**:** Assumptions for instance variables of the class τ.
$\tau.f : \#\,\theta\,(\theta_1, \ldots, \theta_n)$**:** Assumptions for functions of the class τ.

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

The function **TYPE** is given as:

TYPE: `TypeAssumptions` × `class`→ `TypeAssumptions` × `CoercionSet`

$\mathbf{TYPE}(\, \mathit{Ass}, \mathsf{Class}(\, \mathit{cl}, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}, \mathit{ivardecls}\,)\,) =$
let
$\mathit{fdecls} = [\mathsf{Fun}(\, f_1, \mathit{lexpr}_1\,), \ldots, \mathsf{Fun}(\, f_n, \mathit{lexpr}_n\,)]$
$\mathit{ftypeass} = \{\, f_i : a_i \mid a_i \textit{ fresh type variables} \,\}$
$\cup \{\, \mathit{clname.this} : \mathit{clname} \,\}$
$\cup \{\, \textit{functions and instance variables of } \tau' \,\}$
Forall $1 \leqslant i \leqslant n$
$\mathit{Ass}_i = \mathit{Ass} \cup \mathit{ivardecls} \cup \mathit{ftypeass} \cup \{\, \mathit{this} : a_i \,\}$
$(\mathit{resTy}_i, \mathit{CoeS}_i) = \mathbf{TYPEExpr}(\, \mathit{Ass}_i, \mathit{lexpr}_i\,)$
in
$(\{\, (f_i : a_i) \mid 1 \leqslant i \leqslant n \,\},$
$\bigcup_i \mathit{CoeS}_i \cup \{\, (\mathit{resTy}_i \lessdot a_i) \mid 1 \leqslant i \leqslant n \,\})$

Example 2. We consider again the class `Matrix` from Example 1. Now we consider only the untyped function `op`.

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

  op = #{ m -> #{ f -> f(Matrix.this, m) } } }
```

In **TYPE** the function **TYPEExpr** is called with the arguments
$Ass_1 = \{\, \mathsf{op} : a_{\mathtt{op}}, \mathtt{Matrix}.\mathit{this} : \mathtt{Matrix}, \mathit{this} : a_1 \,\}$ and
$\mathsf{Lambda}(\,\mathtt{m}, \mathsf{Lambda}(\,f, \mathsf{Eval}(\,f, \mathtt{Matrix}.\mathit{this}, \mathtt{m}\,)\,)\,)$.

The result is:
$(\{\, \mathsf{op} : a_{op} \,\},$
$\{\, (a_{\#\mathtt{m}} \lessdot a_{\mathtt{op}}), (\#\, a_{\#f}\,(a_m) \lessdot a_{\#\mathtt{m}}), (\#\, a_{f(\,M.this,m\,)}\,(a_f) \lessdot a_{\#f}),$
$(a_f \lessdot \#\, a_3\,(a_1, a_2)), (\mathtt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{f(\,M.this,m\,)}\,\}),$
where the indices of the type variables named by its subterms.

The functions MATCH and SIMPLIFY: The structure of the function **MATCH** is unchanged. Fuh and Mishra's **MATCH** is based on the unfication algorithm of Montanari and Martelli [MM82]. We replace the unfication by our type unification for the Java_λ type system [Plü09]. Furthermore, the set of type constructors is extended. In contrast to Fuh and Mishra, where subtypes are only defined on constants, in Java_λ subtypes on parameterized class-names are allowed as well. During the function **MATCH** coercions are simplified, such that the function **SIMPLIFY** can be integrated into **MATCH**.
Analogously as in [Plü09] we denote $\theta \lessdot \theta'$ for coercions, which should be type unified, which means that there is a substitution σ with $\sigma(\,\theta\,) \leq^* \sigma(\,\theta'\,)$. During the **MATCH** algorithm $\lessdot$ is replaced by $\lessdot_?$ and $\doteq$, respectively. $\theta \lessdot_? \theta'$ means that the two sub-terms θ and θ' of type terms should be matched, such that $\sigma(\,\theta\,)$ is a sub-term subtype of $\sigma(\,\theta'\,)$ (in the sense of the fourth item in the subtyping definition (Def. 3)). $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\,\theta\,) = \sigma(\,\theta'\,)$.
The function **MATCH** determines a minimal matching instance of a given set of coercions. Therefore we have to define a matching predicate and a minimal matching instance for the Java_λ type system.

Definition 5 (matching). *The matching predicate is defined as:*

matching$(\,\theta \lessdot \theta'\,)$,	*if θ and θ' are simple types and there is a substitution σ, such that $\sigma(\,\theta\,) \leq^* \sigma(\,\theta'\,)$.*
matching$(\,\theta \lessdot_? \theta'\,)$,	*if θ and θ' are simple types and there is a substitution σ, such that $\sigma(\,\theta\,)$ is a sub-term subtype of $\sigma(\,\theta'\,)$.*
matching$(\,\theta \doteq \theta'\,)$,	*if θ and θ' are simple types and there is a substitution σ, such that $\sigma(\,\theta\,) = \sigma(\,\theta'\,)$.*
matching$(\,\#\,\theta_0\,(\theta'_1, \ldots, \theta'_n) \lessdot \#\,\theta'_0\,(\theta_1, \ldots, \theta_n)\,)$,	*if matching*$(\,\theta_i \lessdot \theta'_i\,), 0 \leqslant i \leqslant n$

Definition 6 (Minimal matching instance). *Let C be a set of coercions.*

- *C is called* <u>*matching*</u>*, if all coercions of C are matching.*
- *Let σ be a substitution. $\sigma(C)$ is called a* <u>*matching instance*</u> *of C, if $\sigma(C)$ is matching.*
- *$\sigma(C)$ is called a* <u>*minimal matching instance*</u>*, if for all matching instances $\sigma'(C)$ there is a substitution σ'' with $\sigma'' \circ \sigma(C) = \sigma'(C)$.*

The algorithm **MATCH** can be considered as a transformation of the set of coercions C to a substitution σ, such that $\sigma(C)$ is a minimal matching. Besides C and S there is a third data-structure M, which is an equivalence relation of simple types, such that for $\theta \; M \; \theta'$ holds $matching(S(\theta) \, R \, S(\theta'))$, for an $R \in \{ \lessdot, \lessdot_? , \doteq \}$. For the result of the **SIMPLIFY** function a set of atomic coercions AC is also used.
For **MATCH** we need some additional auxillary definitions:

- $[a]_M = \{ a' \mid a \; M \; a' \}$
- $[t]^M = \{ [a]_M \mid a \; occurs \; in \; t \}$
- If A is a set of pairs of simple types, then A^* is its reflexive, symmetric, and transitive closure.
- $ALLNEW(t)$ substitutes in a type t all simple types by fresh type variables.
- $Pair(\# a_0 \, (a'_1, \ldots, a'_n), \# a'_0 \, (a_1, \ldots, a_n)) = \{ (a_i, a'_i) \mid 0 \leqslant i \leqslant n \}$

The function **MATCH** is given as:

MATCH: `CoercionSet` $\rightarrow$`Substitutes` $\times$ `ACoercionSet` $+ \{ fail \}$

MATCH$(C) =$ **MATCH1**$(\emptyset, C, [\,], \{ (\theta, \theta) \mid \theta \; is \; simple \; type \; in \; C \})$

While **MATCH** initializes the data-structures, **MATCH1** determines the result:

$$\mathbf{MATCH1}(AC, C \cup \{ (\# \theta_0 \, (\theta'_1, \ldots, \theta'_n) \lessdot \# \theta'_0 \, (\theta_1, \ldots, \theta_n)) \}, \sigma, M) = \qquad (1)$$
$$\quad \mathbf{MATCH1}(AC, C \cup \{ \theta_i \lessdot \theta'_i \mid 0 \leqslant i \leqslant n \}, \sigma, M)$$

$$\mathbf{MATCH1}(AC, C \cup \{ sty \lessdot sty' \}, \sigma, M) = \qquad (2)$$
$$\quad \textit{where sty and sty' are simple types and sty or sty' is a type variable}$$
$$\quad \mathbf{MATCH1}(AC \cup \{ sty \lessdot sty' \}, C, \sigma, (M \cup \{ sty, sty' \})^*)$$

$$\mathbf{MATCH1}(AC, C \cup \{ (C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot C'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>}) \}, \sigma, M) = \qquad (3)$$
$$\quad \mathbf{let}\; C' = reduce(C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot C'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>})^4$$
$$\quad \mathbf{in}\; \mathbf{MATCH1}(AC \cup C', C, \sigma, (M \cup \{ (\theta, \theta') \mid (\theta \, R \, \theta') \in C' \})^*)$$

$$\mathbf{MATCH1}(AC, C \cup \{ e \}, \sigma, M) = \qquad (4)$$
$$\quad \textit{where either} \;\; e = (v \; R \; \# \theta_0 \, (\theta_1, \ldots, \theta_n))$$
$$\quad \textit{or} \;\; e = (\# \theta_0 \, (\theta_1, \ldots, \theta_n) \; R \; v), R \in \{ \lessdot, \lessdot_? , \doteq \}$$
$$\quad \mathbf{if}\; [v]_M \in [t]^M \;\; \textit{or} \; [v]_M \; \textit{contains a simple type} \; \mathbf{then} \; \textit{fail}$$

---

[4] `reduce` reduces the two Java$_\lambda$ type terms, as in the Java type unification algorithm [Plü09] (step 1, Fig. 1 and 2) described, such that the result consists of pairs, where at least one type term is a type variable.

else let

$t = ALLNEW(\,\#\,\theta_0\,(\theta_1,\ldots,\theta_n)\,)$

$\sigma' = [v \mapsto t]$

$p = PAIR(\,\#\,\theta_0\,(\theta_1,\ldots,\theta_n), t\,)$ (*resp.* $PAIR(\,t, \#\,\theta_0\,(\theta_1,\ldots,\theta_n)\,))$

in

let $(ChAC, UnChAC) = substDevide(\,\sigma', AC\,)$

in MATCH1$(UnChAC \cup \{\,\theta \lessdot \theta' \mid (\theta,\theta') \in p\,\},$

$\sigma'(\,C\,) \cup ChAC,$

$\sigma' \circ \sigma,$

$(M \cup p)^*)$

where substDevide applies σ' to AC and divides $\sigma'(\,AC\,)$ into changed and unchanged coercions.

$$\textbf{MATCH1}(\,AC, \emptyset, \sigma, M\,) = (\sigma, AC) \tag{5}$$

Example 3. We continue Example 2. The set of coercions is given as:
$C = \{\,(a_{\#\mathtt{m}} \lessdot a_{\mathtt{op}}), (\#\,a_{\#f}\,(a_m) \lessdot a_{\#\mathtt{m}}), (\#\,a_{f(\,M.this,m\,)}\,(a_f) \lessdot a_{\#f}),$
$(a_f \lessdot \#\,a_3\,(a_1,a_2)), (\mathtt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{f(\,M.this,m\,)}\,\},$

1. With (2) $AC_1 = \{\,(a_{\#\mathtt{m}} \lessdot a_{\mathtt{op}})\,\}$
2. With (4) follows from $(\#\,a_{\#f}\,(a_m) \lessdot a_{\#\mathtt{m}})$:
   $\sigma' = [a_{\#m} \mapsto \#\,\beta'\,(\beta)]$
   $\sigma_2 = \sigma'$
   $AC_2 = \{\,(a_{\#f} \lessdot \beta'), (\beta \lessdot a_m)\,\}$
3. With (4) follows from $\sigma_2(\,(a_{\#\mathtt{m}} \lessdot a_{\mathtt{op}})\,) = (\#\,\beta'\,(\beta) \lessdot a_{\mathtt{op}})$:
   $\sigma' = [a_{\mathtt{op}} \mapsto \#\,\beta'_1\,(\beta_1)]$
   $\sigma_3 = \sigma' \circ \sigma_2$
   $AC_3 = \{\,(\beta' \lessdot \beta'_1), (\beta_1 \lessdot \beta)\,\}$
4. With (4) follows from $(\#\,a_{f(\,M.this,m\,)}\,(a_f) \lessdot a_{\#f})$:
   $\sigma' = [a_{\#f} \mapsto \#\,\gamma'_1\,(\gamma_1)]$
   $\sigma_4 = \sigma' \circ \sigma_2$
   $AC_4 = \{\,(a_{f(\,M.this,m\,)}, \lessdot\gamma'_1), (\gamma_1 \lessdot a_f)\,\}$
5. With (4) follows from $(a_f \lessdot \#\,a_3\,(a_1,a_2))$:
   $\sigma' = [a_f \mapsto \#\,\epsilon''_1\,(\epsilon_1,\epsilon'_1)]$
   $\sigma_5 = \sigma' \circ \sigma_3$
   $AC_5 = \{\,(\epsilon''_1 \lessdot a_3), (a_1,\epsilon_1), (a_2,\epsilon'_1)\,\}$
6. With (4) follows from $\sigma_5(\,\gamma_1 \lessdot a_f\,) = (\gamma_1 \lessdot \#\,\epsilon''_1\,(\epsilon_1,\epsilon'_1))$:
   $\sigma' = [\gamma_1 \mapsto \#\,\epsilon''_2\,(\epsilon_2,\epsilon'_2)]$
   $\sigma_6 = \sigma' \circ \sigma_5$
   $AC_6 = \{\,(\epsilon''_2 \lessdot \epsilon''_1), (\epsilon_1 \lessdot \epsilon_2)(\epsilon'_1 \lessdot \epsilon'_2)\,\}$
7. With (4) follows from $\sigma_6(\,a_{\#f} \lessdot \beta'\,) = (\#\,\gamma'_1\,(\#\,\epsilon''_2\,(\epsilon_2,\epsilon'_2)) \lessdot \beta')$:
   $\sigma' = [\beta' \mapsto \#\,\gamma'_2\,(\#\,\epsilon''_3\,(\epsilon_3,\epsilon'_3))]$.
   $\sigma_7 = \sigma' \circ \sigma_6$
   $AC_7 = \{\,(\gamma'_1 \lessdot \gamma'_2), (\epsilon''_3 \lessdot \epsilon''_2), (\epsilon_2 \lessdot \epsilon_3)(\epsilon'_2 \lessdot \epsilon'_3)\,\}$

8. With (4) follows from $\sigma_7(\, \beta' \lessdot \beta'_1 \,) = (\#\, \gamma'_2\, (\#\, \epsilon''_3\, (\epsilon_3, \epsilon'_3)) \lessdot \beta'_1 \,)$:
   $\sigma' = [\beta'_1 \mapsto \#\, \gamma'_3\, (\#\, \epsilon''_4\, (\epsilon_4, \epsilon'_4))]$
   $\sigma_8 = \sigma' \circ \sigma_7$
   $AC_8 = \{\, (\gamma'_2 \lessdot \gamma'_3), (\epsilon''_4 \lessdot \epsilon''_3), (\epsilon_3 \lessdot \epsilon_4), (\epsilon'_3 \lessdot \epsilon'_4) \,\}$
9. With (2) follows:
   $AC_9 = \{\, (\texttt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{f(\, M.this, m\,)}) \,\}$

Result: MATCH$(\, C\,) = (\sigma, AC)$, where

$$\begin{aligned}\sigma = \{\, & (a_{\texttt{op}} \mapsto \#\,\#\, \gamma'_3\, (\#\, \epsilon''_4\, (\epsilon_4, \epsilon'_4))\, (\beta_1)),\\ & (a_{\#\texttt{m}} \mapsto \#\,\#\, \gamma'_2\, (\#\, \epsilon''_3\, (\epsilon_3, \epsilon'_3))\, (\beta)),\\ & (a_{\# f} \mapsto \#\, \gamma'_1\, (\#\, \epsilon''_2\, (\epsilon_2, \epsilon'_2))), (a_f \mapsto \#\, \epsilon''_1\, (\epsilon_1, \epsilon'_1)) \,\}\end{aligned}$$

and

$$\begin{aligned}AC = \{\, & \beta \lessdot a_m, \beta \lessdot \beta_1, a_{f(\, this, m\,)} \lessdot \gamma'_1, \epsilon''_1 \lessdot a_3, a_1 \lessdot \epsilon_1, a_2 \lessdot \epsilon'_1,\\ & \epsilon''_2 \lessdot \epsilon''_1, \epsilon_1 \lessdot \epsilon_2, \epsilon'_1 \lessdot \epsilon'_2, \gamma'_1 \lessdot \gamma'_2, \epsilon''_3 \lessdot \epsilon''_2, \epsilon_2 \lessdot \epsilon_3, \epsilon'_2 \lessdot \epsilon'_3,\\ & \gamma'_2 \lessdot \gamma'_3, \epsilon''_4 \lessdot \epsilon''_3, \epsilon_3 \lessdot \epsilon_4, \epsilon'_3 \lessdot \epsilon'_4, \texttt{Matrix} \lessdot a_1, a_m \lessdot a_2, a_3 \lessdot a_{f(\, this, m\,)} \,\}\end{aligned}$$

The function CONSISTENT: The function **CONSISTENT** checks, if for a set of coercions (constraints) a solution exists. The structure of Fuh and Mishra's function **CONSISTENT** is unchanged. Only in the functions $\uparrow$ and $\downarrow$, which determine all supertypes and all subtypes of a set of types, respectively, have to be replaced.
For this we use the functions **greater** and **smaller** from [Plü09].
For a finite set of simple type T holds:

- $T \uparrow = \{\, \theta' \mid \exists\, \theta \in T, \theta' \in \textbf{greater}(\, \theta\,) \,\}$
- $T \downarrow = \{\, \theta \mid \exists\, \theta' \in T, \theta \in \textbf{smaller}(\, \theta'\,) \,\}$
- $T \uparrow_{arg} = \{\, \theta' \mid \exists\, \theta \in T, \theta' \in \textbf{grArg}(\, \theta\,) \,\}$
- $T \downarrow_{arg} = \{\, \theta \mid \exists\, \theta' \in T, \theta \in \textbf{smArg}(\, \theta'\,) \,\}$

The function **greater**$(\, \theta\,)$ and **smaller**$(\, \theta'\,)$ determines a finite set of supertypes of θ respectively a finite set of subtypes of θ', such that all supertypes respectively all subtypes are given as instances of them.
The functions **grArg** and **smArg** determine the supertypes respectively the subtypes of sub-terms, which are allowed as arguments in simple types[5].

COMPRESS$(\, T_1, T_2\,) =$ *if* $T_1 \cap T_2 = \emptyset$ *then fail*
else if $(T_1 \cap T_2) = T_1$ *then* $(True, T_1)$ *else* $(False, T_1 \cap T_2)$

With each simple type θ in the set of coercions we associate a set I_θ, which contains all simple types that can be instantiated to.

The function **CONSISTENT** is given as:

CONSISTENT: `ACoercionSet` $\rightarrow$ `Boolean`

[5] For types without wildcards **greater** determines all supertypes and **smaller** all subtypes, while **grArg** and **smArg** determines no further types.

$\mathbf{CONSISTENT}(\, AC\,) = \mathbf{let}$

$$\forall\, \theta \in AC:\ I_\theta = \begin{cases} \theta \textit{ if } \theta \textit{ is no type variable} \\ * \textit{ if } \theta \textit{ is a type variable} \end{cases}$$

$\mathbf{in}\ \mathbf{CONSISTENT1}(\, \texttt{True}, AC, I\,)$

The function **CONSISTENT1** controls the iteration.

$\mathbf{CONSISTENT1}(\, \mathit{stable}, AC, I\,) =$
$\quad \mathbf{let}\ (\mathit{stable}, I) = \mathbf{CONSISTENT2}(\, \mathit{stable}, AC, I\,)$
$\quad \mathbf{in}\ \mathbf{if}\ \mathit{stable}\ \mathbf{then}\ \texttt{True}$
$\quad \mathbf{else}\ \mathbf{CONSISTENT1}(\, \mathit{stable}, AC, I\,)$

The function **CONSISTENT2** determines the next step:

$\mathbf{CONSISTENT2}(\, \mathit{stable}, AC \cup \{\, \theta\, R\, \theta' \,\}, I\,) =$
$\quad \mathbf{let}$ (stable, I) $=$
$\quad\quad \mathbf{case}\ R\ \mathbf{of}$
$\quad\quad \lessdot$: $\mathbf{let}$
$\quad\quad\quad (\mathit{stable}, I_{\theta'}) = \mathbf{let}\ (\mathit{flag}, \overline{I}) = \mathbf{COMPRESS}(\, I_{\theta'}, I_\theta \uparrow)$
$\quad\quad\quad\quad \mathbf{in}\ (\mathit{stable} \wedge \mathit{flag}, \overline{I})$
$\quad\quad\quad (\mathit{stable}, I_\theta) = \mathbf{let}\ (\mathit{flag}, \overline{I}) = \mathbf{COMPRESS}(\, I_\theta, I_{\theta'} \downarrow)$
$\quad\quad\quad\quad \mathbf{in}\ (\mathit{stable} \wedge \mathit{flag}, \overline{I})$
$\quad\quad\quad \mathbf{in}\ (\mathit{stable}, I \textit{ with substituted } I_\theta \textit{ and } I_{\theta'})$
$\quad\quad \lessdot_?$: $\mathbf{let}$
$\quad\quad\quad (\mathit{stable}, I_{\theta'}) = \mathbf{let}\ (\mathit{flag}, \overline{I}) = \mathbf{COMPRESS}(\, I_{\theta'}, I_\theta \uparrow_{arg})$
$\quad\quad\quad\quad \mathbf{in}\ (\mathit{stable} \wedge \mathit{flag}, \overline{I})$
$\quad\quad\quad (\mathit{stable}, I_\theta) = \mathbf{let}\ (\mathit{flag}, \overline{I}) = \mathbf{COMPRESS}(\, I_\theta, I_{\theta'} \downarrow_{arg})$
$\quad\quad\quad\quad \mathbf{in}\ (\mathit{stable} \wedge \mathit{flag}, \overline{I})$
$\quad\quad\quad \mathbf{in}\ (\mathit{stable}, I \textit{ with substituted } I_\theta \textit{ and } I_{\theta'})$
$\quad\quad \doteq$: (stable, I)
$\quad \mathbf{in}\ \mathbf{CONSISTENT2}($stable, AC, I$)$

$\mathbf{CONSISTENT2}(\, \mathit{stable}, \emptyset, I\,) = (\mathit{stable}, I)$

Now we continue our example.

Example 4. Let C, σ, and

$$\begin{aligned} AC = \{\, & \beta \lessdot a_m, \beta \lessdot \beta_1, a_{f(\,this,m\,)} \lessdot \gamma_1', \epsilon_1'' \lessdot a_3, a_1 \lessdot \epsilon_1, a_2 \lessdot \epsilon_1', \\ & \epsilon_2'' \lessdot \epsilon_1'', \epsilon_1 \lessdot \epsilon_2, \epsilon_1' \lessdot \epsilon_2', \gamma_1' \lessdot \gamma_2', \epsilon_3'' \lessdot \epsilon_2'', \epsilon_2 \lessdot \epsilon_3, \epsilon_2' \lessdot \epsilon_3', \\ & \gamma_2' \lessdot \gamma_3', \epsilon_4'' \lessdot \epsilon_3'', \epsilon_3 \lessdot \epsilon_4, \epsilon_3' \lessdot \epsilon_4', \texttt{Matrix} \lessdot a_1, a_m \lessdot a_2, a_3 \lessdot a_{f(\,this,m\,)} \,\} \end{aligned}$$

from Example 3 be given again.
For each $\theta \lessdot \theta' \in AC$ we have to determine I_θ respectively $I_{\theta'}$. If for all θ, $I_\theta \neq \emptyset$ the atomic coercion set is consistent.

In Table 1 we consider the iteration steps[6].

This means $\mathbf{CONSISTENT}(\, AC\,) = \texttt{true}$.

[6] We consider only the non-wildcard types and abbreviate `Matrix` by `M` and `Vector<Vector<Integer>>` by `V<V<Int>>`.

It	$Coercion$	$I_{\texttt{Matrix}}$	I_{a_1}	I_{ϵ_1}	I_{ϵ_2}	I_{ϵ_3}	I_{ϵ_4}	...
0		M	$*$	$*$	$*$	$*$	$*$	$*$
1	...	M	$*$	$*$	$*$	$*$	$*$	$*$
1	$\texttt{M} \lessdot a_1$	M	M, V<V<Int>>	$*$	$*$	$*$	$*$	$*$
2	...	M	M, V<V<Int>>	$*$	$*$	$*$	$*$	$*$
2	$a_1 \lessdot \epsilon_1$	M	M, V<V<Int>>	M, V<V<Int>>	$*$	$*$	$*$	$*$
2	$\epsilon_1 \lessdot \epsilon_2$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$	$*$	$*$
2	$\epsilon_2 \lessdot \epsilon_3$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$	$*$
2	$\epsilon_3 \lessdot \epsilon_4$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$
2	...	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$
3	...	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$

Table 1. Example CONSISTENT

The algorithm WTYPE: The presented algorithms are now summerized in the adaptation of **WTYPE**. In **WTYPE** an own well-typing is determined for each function declaration.
The function **WTYPE** is given as:

$$\textbf{WTYPE}\colon \texttt{TypeAssumptions} \times \texttt{class} \rightarrow \{\, \texttt{WellTyping} \,\} \cup \{\, fail \,\}$$

$$
\begin{aligned}
&\textbf{WTYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,)\,) = \\
&\quad \textbf{let} \\
&\qquad (\{\, f_1 : a_1, \ldots, f_n : a_n \,\}, CoeS) = \textbf{TYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad fdecls, ivardecls \,)\,) \\
&\qquad (\sigma, AC) = \textbf{MATCH}(\, CoeS \,) \\
&\quad \textbf{in} \\
&\qquad \textbf{if } \textbf{CONSISTENT}(\, AC \,) \textbf{ then} \\
&\qquad\quad \{\, (AC, Ass \vdash f_i : \sigma(\, a_i \,)) \mid 1 \leqslant i \leqslant n \,\} \\
&\qquad \textbf{else } fail
\end{aligned}
$$

Now we will complete our example.

Example 5. In 2 we declared the Java_λ program

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

  op = #{ m -> #{ f -> f(Matrix.this, m) } } },
```

determined the set of assumptions Ass_1 and the type assumptions $\texttt{op} : a_{op}$.
In 3 we determined the substition σ and the set of atomic coercions AC and in 4 we showed that AC is consistent.
From this follows, that in WTYPE for op the well-typing

$$(AC, Ass_1 \vdash \texttt{op} : \#\,\#\,\gamma_3'\,(\#\,\epsilon_4''\,(\epsilon_4, \epsilon_4'))\,(\beta_1)).$$

is determined.
As in the function CONSISTENT all possible instances are determined, it holds: $\epsilon_4 =$ `Matrix` or `Vector<Vector<Integer>>`.
Furthermore, from AC follows, that it holds $\beta_1 \lessdot \epsilon_4'$ and $\epsilon_4'' \lessdot \gamma_3'$, which describes corelations of type variables of the result type.

5 Conclusion and future work

We have considered the Java 8 extensions closures and function types as first-class citizens. We gave an abstract definition of the subtyping relation for a small core language Java_λ. We gave the adaptation of Fuh and Mishra's type inference algorithm [FM88] to Java_λ.
The approach allows to use function types without confusing the programmer, as the function types do not need to be given explictly. This would allow to introduce explicit function types in Java.
In the future we have to develop an IDE for Java_λ. One possibility is to extend the byte-code, such that well-typings are included. The implementation of well-typings could be done similar as the implementation of generics, such that atomic coercions are used for the type check, while the JVM works only with the type `Object`.
The other approach would be to determine for each type variable all possible type combinations by **CONSISTENT** and present the programmer a select box similar as in our implementation of [Plü07]. The programmer selects the favored type, such that a standard typing for the byte-code can be generated.
Furthermore, we want to develop an approach how to use well-typings without type inference. This is important to have a possibility to restrict a type of a function. A first idea could be extending the bounded type variables to atomic coercions.

References

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

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

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

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

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. ACM Transactions on Programming Languages and Systems, 4:258–282, 1982.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, 5th International Conference on Principles and Practices of Programming in Java, volume 272 of ACM International Conference Proceeding Series, pages 73–82, September 2007.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin 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.