Functional implementation of well-typings in Java_λ

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 λ–expressions. We have extended a subset of Java 7 by λ–expressions and function types. We call this language Java_λ. For Java_λ we presented a type inference algorithm. In this contribution we present a prototypical implementation of the type inference algorithm implemented in Haskell.

1 Introduction

In the late eighties Fuh and Mishra have presented a type inference algorithm for a small function programming language with subtyping and without overloading [FM88].
In Java_λ we have a similiar situation. Subtyping is allowed and functions, which are declared by λ–expressions, are not overloaded.
We have adapted the Fuh and Mishra algorithm to a type inference algorithm for Java_λ [Plü11]. The main difference is the definition of the subtyping ordering. Therefore follows that the unification in [FM88] had to be substituted by our type unification [Plü09].
The type inference algorithm consists of three functions:

TYPE: The function **TYPE** types each sub-term of the λ–expressions by type variables and determines corecions (subtype pairs), which have to be solved.
MATCH: The function **MATCH** adapts the structure of the types of each subtype pair and reduces the coercions to atomic coercions. An atomic coercion is a subtype pair, where the types consists only of type variables and type constants.
CONSISTENT: The function **CONSISTENT** determines iteratively solutions for the atomic coercions. If there is at least one solution, the result is true. This means that there is a correct typing for the λ–expressions. Otherwise, the algorithm fails.

The algorithm itself is given as:

WTYPE: $\texttt{TypeAssumptions} \times \texttt{class} \rightarrow \{\, \texttt{WellTyping} \,\} \cup \{\, \mathit{fail} \,\}$

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

The result of the algorithm is the set of well-typings:

$$\{\,(AC, Ass \vdash f_i : \sigma(\,a_i\,)) \mid 1 \leqslant i \leqslant n\,\}$$

where

- AC is a set of coercions,
- Ass is a set of type assumptions,
- f_i are function names, and
- $\sigma(\,a_i\,)$ are types.

It is a problem that well-typings are not included in the Java type-system.
If we consider **CONSISTENT** more detailed, we will recognize, that for all types, which are in relation with a non-variable type, all possible instances are determined. We call a function, which gives these instances as result, **SOLUTIONS**. This means that the set of corecions could be reduced to a set AC' consisting only type variables. These pairs could be expressed by bounded type variables in Java. Here is a small extention necessary, e.g. parameters of a function could also be a bound of another parameter.
Hence the algorithm looks like this:

$\mathbf{WTYPE}$: $\texttt{TypeAssumptions} \times \texttt{class} \rightarrow \{\,\texttt{WellTyping}\,\} \cup \{\,fail\,\}$

$\mathbf{WTYPE}(\,Ass, \mathsf{Class}(\,cl, \mathsf{extends}(\,\tau'\,), fdecls, ivardecls\,)\,) =$
$\quad\mathbf{let}$
$\qquad(\{\,f_1 : a_1, \ldots, f_n : a_n\,\}, CoeS) =$
$\qquad\quad\mathbf{TYPE}(\,Ass, \mathsf{Class}(\,cl, \mathsf{extends}(\,\tau'\,), fdecls, ivardecls\,)\,)$
$\qquad(\sigma, AC) = \mathbf{match}(\,CoeS\,)$
$\qquad((\tau_1, \ldots, \tau_m), AC') = \mathbf{SOLUTIONS}(\,AC\,)$
$\quad\mathbf{in}$
$\qquad\{\,(AC', Ass \vdash \{\,f_i : \tau_j \circ \sigma(\,a_i\,) \mid 1 \leqslant i \leqslant n\,\}) \mid 1 \leqslant j \leqslant m\,\}$

2 The language

The language Java_λ is an extension of our language in [Plü07] by λ–expressions and function types. Java_λ is the core of the language, which is described by

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

Fig. 1. The abstract syntax of Java_λ

Reinhold's in [lam10]. In (Fig. 1) an abstract representation is given, where the additional features are underlined. Beside instance variables functions can be declared in classes. A function is declared by its name, optionally its type, and a 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.

Definition 1 (Types). *Let Ty_p be a set of Java 5.0 types ([GJSB05], Section 4.5), where BTV is an indexed set of bounded type variables. Then the set of Java_λ types $\mathsf{Type}_{TS}(\,BTV\,)$ is defined by*

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

$$\#\, ty\,(ty_1, \ldots, ty_n) \in \mathsf{Type}_{TS}(\,BTV\,)^1$$

Example 1. We consider the class `Matrix`.

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

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

}
```

[1] Often function types $\#\, ty\,(ty_1, \ldots, ty_n)$ are written as $(ty_1, \ldots, ty_n) \ \rightarrow\ ty$.

`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. The function `op` is untyped. The first argument `m` and the second argument `f` are also untyped. The first idea for a correct typing could be that `m` gets the type `Matrix` and `f` gets $\#\,\texttt{Matrix}\,(\texttt{Matrix}, \texttt{Matrix})$, which mean that the function `f` has the type $\#\,\#\,\texttt{Matrix}\,(\texttt{Matrix}, \texttt{Matrix})\,(\texttt{Matrix})$.

The main difference between Java_λ and the corresponding core of Java 8 [Goe11] is the typing of λ–expressions. While in Java 8 the types are given as functional interfaces (Java interfaces with one method) in Java_λ the types of λ–expressions are given as real function types.

3 Implementation

In the following context it is described how to implement the algorithm **WTYPE** in Haskell. The background was explained in the introduction (Section 1). The algorithm for the Java_λ itself is given in [Plü11].

3.1 Abstract syntax

The data-structure for a class is given as

```
data Class = Class(SType, --name
                   [SType], -- extends
                   [IVarDecl], -- instancevariables
                   [FunDecl]) -- functiondeclarations
```

The first argument is the class-name, the second argument the super-class, respectively the implemented interfaces, the third argument the list of instancs variables, and the fourth argument the function declarations.

```
data FunDecl = Fun(String, Maybe Type, Expr)
```

A function is declared by its name, an optionally type and an expression. The optionally type will be inferred by the type-inference algorithm.
We consider only the new construtions of the data-structures `Expr` for expressions and `StmtExpr` for statement-expressions. The data-structure `Stmt` for statements is unchanged.

```
data Expr = Lambda([Expr], Lambdabody)
          | InstFun(Expr, String, String)
          | ...

data Lambdabody = StmtLB(Stmt)
                | ExprLB(Expr)
```

An expressions could be a λ–expression, the first argument is a list of parameters and the second argument, the λ–body, is either a statement or an expression.
The other considered constructor is the instance of a function. The first argument is the expression, which represents the class-instance, which comprises the function. The second argument represents the class name. This is necessary, as the algorithm allows no overloading. The third argument finally is the function name.

```
data StmtExpr = Eval(Expr, [Expr])
              | ...
```

The first argument of the constructor `Eval` is an expression, which represents a function. The second argument is a list of arguments. `Eval` stands for the evaluation of the functions application to the arguments.

Example 2. The abstract syntax of the class `Matrix` (Example 1) is given as:

```
[Class(TC ("Matrix",[]),
       [TC ("Vector",[TC ("Vector",[TC ("Integer",[])])])],
       [],
       [Fun(
         "op",
         Nothing,
         Lambda([LocalOrFieldVar "m",],
                ExprLB(
                  Lambda([LocalOrFieldVar "f"],
                         ExprLB(StmtExprExpr(
                           Eval(LocalOrFieldVar "f",
                                [ThisStype "Matrix",
                                 LocalOrFieldVar "m"]))))))))])]
```

3.2 Parser

The parser is defined by a HAPPY–File. HAPPY is the LR-parser-generating–tool of Haskell. The syntax is similar to yacc. In Figure 2 a part of the specification is given.
Against to yacc in HAPPY the commands of the rules are given as return-expressions. This means that no `$$` is necessary to return a value.
The function `divideFuncInstVar` divides declarations of instance-variables and functions, as in Java mixed declarations are allowed.
`FType` is the constructor for the function type, the representation of $\#\, rettype\, (argtypes)$.
`TypeSType` is the boxed representation of Java 5.0 types in the set of all types.

```
classdeclaration : CLASS IDENTIFIER classpara classbody
                   { Class(TC($2, $3), [], fst $4, snd $4) }

classbody        : LBRACKET RBRACKET  { ([], []) }
                 | LBRACKET classbodydeclarations RBRACKET
                   { divideFuncInstVar $2 ([], []) }

fundeclaration   : funtype IDENTIFIER ASSIGN expression SEMICOLON
                   { Fun($2, Just $1, $4)}
                 | IDENTIFIER ASSIGN expression SEMICOLON
                   { Fun($1, Nothing, $3)}

funtype          : SHARP funtypeortype LBRACE funtypelist RBRACE
                   { FType($2, $4) }

funtypeortype    : funtype { $1 }
                 | type { TypeSType $1 }
```

Fig. 2. Part of the Java$_\lambda$ HAPPY–File

3.3 The function TYPE

The function **TYPE** introduces fresh type variables to each sub-term of the expressions and determines the coercions (subtype pairs). The function needs a set of type-assumptions and a unique number for the next fresh type variable. We encapsulate these in a monad (Figure 3).

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    ...

data M a = Mon((TypeAssumptions, Int) -> (a, (TypeAssumptions, Int)))

instance Monad (M) where
    return coe_lexpr = Mon(\ta_nr -> (coe_expr, ta_nr))
    (>>=) (Mon f1) f2 = Mon (\ta_nr ->
                             let (coe_lexpr, ta_nr') = f1 ta_nr
                             in getCont(f2 coe_lexpr) ta_nr')

getCont:: M a -> ((TypeAssumptions, Int) -> (a, (TypeAssumptions, Int)))
getCont (Mon f) = f
```

Fig. 3. Monad for the function **TYPE**

The function `return` encapsulates a pair (coercion set, expression) to a functions which takes a pair (type-assumptions, number) and returns the pair of pairs ((coercion set, expression), (type-assumptions, number)).

The function `>>=` (bind-operator) takes an encapsulated function and another function. The result is the encapsulated function, which concatenates both functions. The function `getCont` decapsulate the content of the monad.
For expressions, statements and statement-expressions in each case a function is needed, which takes an expression, a statement, or a statement-expression, respectively and returns a corresponding monad. A Java-program consists of different functions, which are declared by (λ–)expressions. Therefore an additional function is necessary, which filters the expressions and calls the TYPE–function.
In Figure 4 a part is presented. The main principle of monadic application is

```
tYPEClass :: Class -> M (CoercionSet, Class)
tYPEClass (Class(this_type, extends, instvar, funs)) =
  let
    funexprlist = map (\(Fun(op, typ, lambdat) -> lambdat) funs
  in
    tYPEExprList funexprlist

tYPEExprList :: [Expr] -> M (CoercionSet, [Expr])
tYPEExprList (e : es) = (tYPEExpr e)
   >>= (\coe_lexpr1 -> (tYPEExprList es)
      >>= \coe_lexp2 ->
          return ((fst coe_lexp1) ++ (fst coe_lexp2),
                  (snd coe_lexp1) : (snd coe_lexp2)))
tYPEExprList [] = return ([], [])

tYPEExpr :: Expr -> M (CoercionSet, Expr)

...

tYPEStmtExpr :: StmtExpr -> M (CoercionSet, StmtExpr)

...

tYPEStmt :: Stmt -> M(CoercionSet, Stmt)
...
```

Fig. 4. The **TYPE**–function

shown in function `tYPEExprList`. First `tYPEExpr` is applied to the first expression. By the bind-operator `>>=` the result is introduced in the recursive call of `tYPEExprList`. Finally, the results of both are summarized in the result of the whole function by dividing the corecions and the typed expressions.

Example 3. If we apply `tYPEClass` to the class `Matrix` (Example 1), we get the set of coercions:

```
([(FType (TypeSType (TFresh "V3"),[TypeSType (TFresh "V2")]),
```

```
   TypeSType(TFresh "V1")),
  (FType (TypeSType (TFresh "V8"),[TypeSType (TFresh "V4")]),
   TypeSType (TFresh "V3")),
  (TypeSType (TFresh "V4"),
   FType(TypeSType (TFresh "V7"),
         [TypeSType(TFresh "V6"),TypeSType (TFresh "V5")])),
  (TypeSType (TFresh "V7"),TypeSType (TFresh "V8")),
  (TypeSType (TC ("Matrix",[])),TypeSType (TFresh "V6")),
  (TypeSType (TFresh "V2"),TypeSType (TFresh "V5"))]
```

and the typed class

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

  V1 op = # { (V2 m) -> # { (V4 f) -> (f).(Matrix.this, m) } };

}
```

In the abstract representation all typed sub-terms could be considered.

```
[Class(TC ("Matrix",[]),
       [TC ("Vector",[TC ("Vector",[TC ("Integer",[])])])],
       [],
       [Fun(
         "op",
         Just(TypeSType (TFresh "V1")),
         TypedExpr(
           Lambda([TypedExpr(LocalOrFieldVar "m",
                             TypeSType (TFresh "V2"))],
                  ExprLB(TypedExpr(
                    Lambda(
                      [TypedExpr(LocalOrFieldVar "f",
                                 TypeSType(TFresh "V4"))],
                      ExprLB(TypedExpr(StmtExprExpr(
                        TypedStmtExpr(
                          Eval(TypedExpr(LocalOrFieldVar "f",
                                         TypeSType(TFresh "V4")),
                               [TypedExpr(ThisStype "Matrix",
                                          TypeSType(TC ("Matrix",[]))),
                                TypedExpr(LocalOrFieldVar "m",
                                          TypeSType(TFresh "V2"))]),
                          TypeSType(TFresh "V8"))),
                      TypeSType(TFresh "V8")))),
                    TypeSType(TFresh "V3")))),
           TypeSType(TFresh "V1")))])]
```

3.4 The function MATCH

The function **MATCH** unifies the coercions and reduces them. The result is a substitution and a set of atomic (reduced) coercions. Atomic coercions consist of pairs of Java 5.0 types.

While in the original algorithm of Fuh and Mishra the ordinary unification is used, for Java$_\lambda$ our type unification [Plü09] is necessary. Our type unification processes also wildcard types.
In Figure 5 the data-structures of MATCH are presented. The type Subst repre-

```
type Subst = [(Type, Type)]
type EquiTypes = [[Type]]
data Rel = Kl | Kl_QM | Eq | Gr | Gr_QM
type CoercionSetMatch = [(Type, Rel, Type)]

--Monade
data M a = Mon((EquiTypes, Int) -> (a, (EquiTypes, Int)))
instance Monad (M) where
    return subst_aCoes = Mon(\eq_nr -> (subst_aCoes, eq_nr))
    (>>=) (Mon f1) f2 = Mon (\eq_nr ->
                             let (subst_aCoes, eq_nr') = f1 eq_nr
                             in getCont(f2 subst_aCoes) eq_nr')
getCont:: M a -> ((EquiTypes, Int) -> (a, (EquiTypes, Int)))
getCont (Mon f) = f
```

Fig. 5. Data-structure of **MATCH**

sents the substitution. The type EquiTypes is necessary for Java 5.0 types which can be considered as equivalent. Rel are the different relations, which are used. QM stands for question mark, the wildcard type in Java.
In the algorithm again a monad is used. (EquiTypes, Int) is the pair of the equivalent types and the number of the next fresh type variable. subst_aCoes is the result, a substitution and a set of atomic coercions.

The algorithm itself consists of five cases:

```
mATCH :: CoercionSetMatch -> CoercionSetMatch -> FC
         -> M(Subst, CoercionSetMatch)

-- decomposition
mATCH aCoes ((FType(ret1, args1), Kl, FType(ret2, args2)):coes) fc = ...

-- reduce
mATCH aCoes ((TypeSType(TC(n1, args1)),rel,
             TypeSType(TC(n2, args2))):coes) fc = ...

-- expansion
mATCH aCoes ((TypeSType(TFresh(name)), rel,
             FType(ret2, args2)) : coes) fc = ...

-- atomic elimination
mATCH aCoes (((TypeSType(TFresh(name))), rel,
```

```
            javafivetype) : coes) fc = ...
```

```
-- recursion base
mATCH aCoes [] fc = (return ([], aCoes))
```

Decomposition: The function-type constructor is erased and the arguments respectively the result types are identified.
Reduce: The type-constructors `n1` and `n2` are reduced.
Expansion: The fresh type variable `name` is expanded, such that the type can be unfied with the function type on the right hand side.
Atomic elimination: The types are introduced in the set of equivalent types.

`FC` respresents the finite closure of the extends-relation [Plü07].

Example 4. `mATCH` applied to the coercions of Example 3 gives:

The substitution:

```
[(TypeSType (TFresh "V14") ↦
  FType(TypeSType (TFresh "V21"),
        [TypeSType (TFresh "V22"),TypeSType (TFresh "V23")])),
 (TypeSType (TFresh "V12") ↦
  FType(TypeSType (TFresh "V18"),
        [TypeSType (TFresh "V19"),TypeSType (TFresh "V20")])),
 (TypeSType (TFresh "V4") ↦
  FType(TypeSType (TFresh "V15"),
        [TypeSType (TFresh "V16"),TypeSType (TFresh "V17")])),
 (TypeSType (TFresh "V9") ↦
  FType (TypeSType (TFresh "V13"),
         [FType (TypeSType (TFresh "V21"),
                 [TypeSType (TFresh "V22"),TypeSType (TFresh "V23")])])),
 (TypeSType (TFresh "V3") ↦
  FType (TypeSType (TFresh "V11"),
         [FType (TypeSType (TFresh "V18"),
                 [TypeSType (TFresh "V19"),TypeSType (TFresh "V20")])])),
 (TypeSType (TFresh "V1") ↦
  FType (FType (TypeSType (TFresh "V13"),
         [FType (TypeSType (TFresh "V21"),
                 [TypeSType (TFresh "V22"),TypeSType (TFresh
                 "V23")])]),
         [TypeSType (TFresh "V10")]))],
```

If we apply the substitution to the typed in the typed program `Matrix`, we get:

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

  ##V13(#V21(V22, V23))(V10)
    op = # { (V2 m) -> # { (#V15(V16, V17) f) -> (f).(Matrix.this, m) } };

}
```

The set of atomic coercions:

```
[(TypeSType (TFresh "V2"),Kl,TypeSType (TFresh "V5")),
 (TypeSType (TC ("Matrix",[])),Kl,TypeSType (TFresh "V6")),
 (TypeSType (TFresh "V7"),Kl,TypeSType (TFresh "V8")),
 (TypeSType (TFresh "V21"),Kl,TypeSType (TFresh "V18")),
 (TypeSType (TFresh "V19"),Kl,TypeSType (TFresh "V22")),
 (TypeSType (TFresh "V20"),Kl,TypeSType (TFresh "V23")),
 (TypeSType (TFresh "V18"),Kl,TypeSType (TFresh "V15")),
 (TypeSType (TFresh "V16"),Kl,TypeSType (TFresh "V19")),
 (TypeSType (TFresh "V17"),Kl,TypeSType (TFresh "V20")),
 (TypeSType (TFresh "V15"),Kl,TypeSType (TFresh "V7")),
 (TypeSType (TFresh "V6"),Kl,TypeSType (TFresh "V16")),
 (TypeSType (TFresh "V5"),Kl,TypeSType (TFresh "V17")),
 (TypeSType (TFresh "V11"),Kl,TypeSType (TFresh "V13")),
 (TypeSType (TFresh "V8"),Kl,TypeSType (TFresh "V11")),
 (TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V2"))])
```

3.5 The function SOLUTIONS

The function CONSISTENT in the original algorithm is in our approach substituted by the function SOLUTIONS. CONSISTENT determines iteratively all possible solutions until it is obvious, that there is a solution. The result is then true, otherwise false. We extend this algorithm such that all possible solutions are determined.

```
sOLUTIONS :: [(Type, Rel, Type)] -> FC -> [[(Type, Type)]]
```

The input is the set of atomic corecions and the finite closure of the extends-relation. The result is the list of correct substitutions.
The algorithm itself has two phases. First all type variables are initialized by '*'. Then in some iterations steps over all coercions all correct instatiations are determined. The result is a list of substitutions, where all type variables, which are not in relation to a non-variable type, are remained instantiated by '*'. These variables can be instantiated by any type, only constraints are given by the coercions.

Example 5. The completion of the `Matrix` example is given by the application of `sOLUTIONS` to the result of `mATCH` (Example 4). There are three different solutions. Applied to the typed program we get:

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

##V13(#V21(Matrix, V23))(V10)
  op = #{ (V2 m) -> #{ (#V15(Matrix,V17) f) -> (f).(Matrix.this,m) } };

}

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

```
  ##V13(#V21(Vector<Vector<Integer>>, V23))(V10)
    op = #{ (V2 m) -> #{ (#V15(Matrix, V17) f) -> (f).(Matrix.this, m) } };

}

class Matrix extends Vector<Vector<Integer>> {

 ##V13(#V21(Vector<Vector<Integer>>, V23))(V10)
  op = #{ (V2 m) ->
         #{ (#V15(Vector<Vector<Integer>>, V17) f) -> (f).(Matrix.this, m) } };

}
```

The type variables `V13`, `V21`, `V23`, `V10`, `V2`, `V15`, and `V17` are not in relation to a non-variable type. This means that these types can be instantiated by an type, but there are coercions, which contrains the possible instatiations. E.g.

```
(TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V23"))
(TypeSType (TFresh "V21"),Kl,TypeSType (TFresh "V13"))
(TypeSType (TFresh "V2"),Kl,TypeSType (TFresh "V17"))
(TypeSType (TFresh "V15"),Kl,TypeSType (TFresh "V13"))
(TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V2"))
```

If we compare this result with the assuption in Example 1, we recognize, that this result is more principal. On the one hand the type of `m` is a type variable and on the other hand the first argument of `f` could be `Matrix` and `Vector<Vector<Integer>>`.

4 Conclusion and Future Work

In this paper we presented the implementation of the adapted Fuh and Mishra's type inference algorithm **WTYPE** to Java_λ. We gave the implementation in Haskell. We presented the parser done by the generating tool HAPPY and the functions **TYPE**, **MATCH**, and **SOLUTIONS**, where **TYPE** and **MATCH** are implementated by a state monad.The result is a well-typing. Well-typings are unknown in Java so far. Constrains of type variables, as the corecions in our approach, can be given in Java by bounds of parameters of classes and functions. A bound can only be a non-variable type. This means to introduce well-typings in the Java type system, the concept of bounds should be extended.
Finally, we show, how the `Matrix` example could be implemented, with extended bounds.

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

  <V10 extends V23, V21 extends V13, V2 extends V17, V15 extends V13,
   V10 extends V2, V23, V13, V17>
  ##V13(#V21(Matrix, V23))(V10)
    op = #{ (V2 m) -> #{ (#V15(Matrix, V17) f) -> (f).(Matrix.this, m) } };
```

}

References

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

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

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

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

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

[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.

[Plü11] Martin Plümicke. Well-typings for Java$_\lambda$. In Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ '11, pages 91–100, New York, NY, USA, 2011. ACM.