Implementation of well-typings in Java$_\lambda$

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

September, 26th 2011

Overview

Function type declaration in Java 7/8

In Mark Reinhold: Project Lambda[1] Java Language Specification draft (Version 0.1.5) explicite function types had been introduced[2]:

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

[1] http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt

[2] At the moment Sun would avoid function types by using SAM–Types

Function type declaration in Java 7/8

In Mark Reinhold: Project Lambda[1] Java Language Specification draft (Version 0.1.5) explicite function types had been introduced[2]:

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

Example:

```
int doOperation(#int(int, int) o, int a, int b)
  {
    return o(a,b);
  }
```

[1] http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt

[2] At the moment Sun would avoid function types by using SAM–Types

Example

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

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

Example

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

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

class Main {
  public static void main(String[] args) {
      Matrix m1 = new Matrix(...);
      Matrix m2 = new Matrix(...);
      m1.op.(m2).(#{ (Matrix m1, Matrix m2) ->
                        Matrix ret = new Matrix ();
                        :
                        : //matrix multiplication
                        return ret;
                   }) } }
```

Goal

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

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

}
```

The algorithm I (Adapt Fuh and Mishra's algorithm[3])

WTYPE : $\texttt{TypeAssumptions} \times \texttt{class} \longrightarrow \{\,\texttt{WellTyping}\,\} + \{\,\textit{fail}\,\}$

Input:

- a set of type assumptions
- a Java$_\lambda$ class (without type annotations)

Output:

- set of well-typings for the functions of the input class

[3] [Pluemicke 2011]: *Well-typings for Java$_\lambda$* , PPPJ 2011

The algorithm I (Adapt Fuh and Mishra's algorithm[3])

WTYPE : $\texttt{TypeAssumptions} \times \texttt{class} \longrightarrow \{\,\texttt{WellTyping}\,\} + \{\,\textit{fail}\,\}$

Input:

- a set of type assumptions
- a Java$_\lambda$ class (without type annotations)

Output:

- set of well-typings for the functions of the input class Well-Typing:

$$(C, A) \vdash N : t$$

  - C = set of coercions (set of sub-type pairs)
  - A = set of type assumptions
  - N = expression
  - t = type

[3] [Pluemicke 2011]: *Well-typings for Java$_\lambda$* , PPPJ 2011

The algorithm II: The sub-function **tYPE**

$$\textbf{tYPE} : \texttt{TypeAssumptions} \times \texttt{class} \longrightarrow \texttt{TypeAssumptions} \times \texttt{CoercionSet}$$

- maps a fresh type variable to each subterm of the functions
- determines a result type (variable) for each function
- determines the corresponding coercions (sub-type pairs)

The algorithm III: The sub-function **match**

match : $\texttt{CoercionSet} \longrightarrow \texttt{Substitution} \times \texttt{ACoercionSet} + \{\, \mathit{fail} \,\}$

- Type unification [Pluemicke 2009][4] to adapt the structure of the coercions.
- Reduce the coercions to atomic coercions (eliminate type constructors)

[4] [Pluemicke 2009]: *Java type unification with wildcards*, INAP 07

The algorithm III: The sub-function **consistent**

consistent : `AtomicCoercionSet` $\longrightarrow$ `Boolean`

- Consistence check of the atomic coercions by intersection set constructions for all possible instatiations
- If the intersection sets are non-empty then the set of atomic coercions is consistent.

The algorithm IV

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

WTYPE(Ass, Class(cl, extends(τ'), fdecls, $\mathit{ivardecls}$)) =
let
$(\{\, f_1 : a_1, \ldots, f_n : a_n \,\}, \mathit{CoeS}) =$
tYPE(Ass, Class(cl, extends(τ'), fdecls, $\mathit{ivardecls}$))
$(\sigma, \mathit{AC}) =$ **match**(CoeS)
in
if consistent(AC) **then**
$\{\, (\mathit{AC}, \mathit{Ass} \vdash f_i : \sigma(\, a_i \,)) \mid 1 \leqslant i \leqslant n \,\}$
else *fail*

The algorithm IV

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

$$
\begin{aligned}
&\textbf{WTYPE}(\, \mathit{Ass}, \mathrm{Class}(\, \mathit{cl}, \mathrm{extends}(\, \tau' \,), \mathit{fdecls}, \mathit{ivardecls} \,) \,) = \\
&\quad \textbf{let} \\
&\qquad (\{\, f_1 : a_1, \ldots, f_n : a_n \,\}, \mathit{CoeS}) = \\
&\qquad\quad \textbf{tYPE}(\, \mathit{Ass}, \mathrm{Class}(\, \mathit{cl}, \mathrm{extends}(\, \tau' \,), \mathit{fdecls}, \mathit{ivardecls} \,) \,) \\
&\qquad (\sigma, \mathit{AC}) = \textbf{match}(\, \mathit{CoeS} \,) \\
&\qquad ((\tau_1, \ldots, \tau_m), \mathit{AC}') = \textbf{solutions}(\, \mathit{AC} \,) \\
&\quad \textbf{in} \\
&\qquad \{\, (\mathit{AC}', \mathit{Ass} \vdash \{\, f_i : \tau_j \circ \sigma(\, a_i \,) \mid 1 \leqslant i \leqslant n \,\}) \mid 1 \leqslant j \leqslant m \,\}
\end{aligned}
$$

Overview

- Abstract syntax
- Parser (HAPPY–Praser generator)
- Function `tYPE`
- Function `match`
- Function `consistent`
- Function `solution`

Abstract syntax

```
data Class = Class(SType, --name
                   [SType], -- extends
                   [IVarDecl], -- Instancevariables
                   [FunDecl]) -- Functiondeclarations
 ...
data FunDecl = Fun(String, Maybe Type, Expr)

data Stmt = ...

data StmtExpr = ...

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

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

HAPPY–File

```
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 }
```

tYPE (Zustandsmonade)

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

tYPE (Zustandsmonade)

```
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')
```

tYPE (Zustandsmonade)

```
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
```

tYPE (Funktionen)

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

tYPE (Funktionen)

```
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)
   >>= (\v1 -> (tYPEExprList es)
      >>= \v2 -> return ((fst v1) ++ (fst v2), (snd v1) : (snd v2)))
tYPEExprList [] = return ([], [])
```

tYPE (Funktionen)

```
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)
   >>= (\v1 -> (tYPEExprList es)
      >>= \v2 -> return ((fst v1) ++ (fst v2), (snd v1) : (snd v2)))
tYPEExprList [] = return ([], [])

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

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

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

Example

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

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

}
```

Example

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

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

}
```

```
tYPEClass "Matrix.java"
```

Datatypes for match

```
type Subst = [(Type, Type)]

type EquiTypes = [[Type]]

data Rel = Kl | Kl_QM | Eq | Gr | Gr_QM

type CoercionSetMatch = [(Type, Rel, Type)]
```

Datatypes for match

```
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
```

match

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

match

```
match :: CoercionSetMatch -> CoercionSetMatch -> FC
                                    -> M(Subst, CoercionSetMatch)
match aCoes
      ((FType(ret1, args1), Kl, FType(ret2, args2)) : coes)
      fc = ...     --  decomposition
```

match

```
match :: CoercionSetMatch -> CoercionSetMatch -> FC
                                    -> M(Subst, CoercionSetMatch)
match aCoes
      ((FType(ret1, args1), Kl, FType(ret2, args2)) : coes)
      fc = ...     --  decomposition

match aCoes
      ((TypeSType(TC(n1, a1)),rel,TypeSType(TC(n2, a))):coes)
      fc = ...     --  reduce
```

match

```
match :: CoercionSetMatch -> CoercionSetMatch -> FC
                                  -> M(Subst, CoercionSetMatch)
match aCoes
      ((FType(ret1, args1), Kl, FType(ret2, args2)) : coes)
      fc = ...     --  decomposition

match aCoes
      ((TypeSType(TC(n1, a1)),rel,TypeSType(TC(n2, a))):coes)
      fc = ...     --  reduce

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

match

```
match :: CoercionSetMatch -> CoercionSetMatch -> FC
                                  -> M(Subst, CoercionSetMatch)
match aCoes
      ((FType(ret1, args1), Kl, FType(ret2, args2)) : coes)
      fc = ...     --  decomposition

match aCoes
      ((TypeSType(TC(n1, a1)),rel,TypeSType(TC(n2, a))):coes)
      fc = ...     --  reduce

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

match aCoes
      (((TypeSType(TFresh(name))), rel, simpletype) : coes)
      fc = ...   --  atomic elimination
```

match

```
match :: CoercionSetMatch -> CoercionSetMatch -> FC
                                    -> M(Subst, CoercionSetMatch)
match aCoes
      ((FType(ret1, args1), Kl, FType(ret2, args2)) : coes)
      fc = ...     --  decomposition

match aCoes
      ((TypeSType(TC(n1, a1)),rel,TypeSType(TC(n2, a))):coes)
      fc = ...     --  reduce

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

match aCoes
      (((TypeSType(TFresh(name))), rel, simpletype) : coes)
      fc = ...   --  atomic elimination

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

Example

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

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

}
```

Example

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

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

}
```

```
match (tYPEClasses "Matrix.java")
```

solutions

```
solutions :: [(SType, Rel, SType)] -> FC -> [[(SType, SType)]]
```

Input: Set of coercions (sub-type pairs)
Finite Closure `FC` of the extends-relation

Output: List of substitutions of solutions

Idea of solutions

```
solutions :: [(SType, Rel, SType)] -> FC -> [[(SType, SType)]]
```

Init: mapping of '*' to each type variable

Iteration: Foreach coercion determination of all possible instantiations.

Example

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

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

}
```

Example

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

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

}
```

```
solutions(match (tYPEClasses "Matrix.java"))
```

Coercions as bounded type variables

```
class Matrix {

  <B1 extends E4', E4', E4'' extends G3', G3', E4 super Matrix>
  ##G3'(#E4''(E4, E4'))(B1)
  op = #{ B1 m -> #{ #E4''(E4, E4') f -> f(Matrix.this, m) }

}
```

This declaration is not allowed in Java.

Conclusion and Future work

Conclusion

- Fuh and Mishra's type inference algorithm can be adopted to Java$_\lambda$.
- Proto-type implementation is done in Haskell.

Conclusion and Future work

Conclusion

- Fuh and Mishra's type inference algorithm can be adopted to Java$_\lambda$.
- Proto-type implementation is done in Haskell.

Future work

- Implementation of the intersection-types by an IDE
- Extension of bounded type variabales
- Integration of SAM–types (Java 8)
- Overloading