Functional implementation of well-typings in Java$_\lambda$

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

September, 1st 2012

Overview

Introduction

Functional features went into Java

- PIZZA (ADT, parametric polymorhism, higher–order functions) [1997]
- GJ (parametric polymorhism) [1998]
- Java 5 (generics, wildcard types (restricted form of existential types) [2005]
- Java 8 (closures/λ–expressions) [2012/13]

Type–inference?

- only local type–inference

The language Java$_\lambda$

Java is extended by

λ–expressions:

```
#{(int x, int y) -> x*y }
```

Functions types:

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

Evaluation:

```
#{int x -> (int y -> x*y) }.(2).(3)
```

The syntax is oriented at Mark Reinhold: *Project Lambda*[1] Java Language Specification draft (Version 0.1.5) (2010)[2]

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

[2] At the moment Oracle would avoid function types by using Functional-interfaces

Type inference

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

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

Type inference

```
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 base: Type-inference algorithm of Fuh and Mishra[2]

WTYPE : $\texttt{TypeAssumptions} \times \texttt{Expr} \rightarrow \texttt{WellTyping} + \{ \mathit{fail} \}$

Input:
- a set of type assumptions
- an expression (Language: λ-expressions with subtyping)

Output:
- a well-typing for the input expression

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] [Fuh/Mishra 1988]: *Type inference with subtypes*, ESOP 1988

The base: Type-inference algorithm of Fuh and Mishra[2]

WTYPE : $\texttt{TypeAssumptions} \times \texttt{Expr} \rightarrow \texttt{WellTyping} + \{ \mathit{fail} \}$

Input:
- a set of type assumptions
- an expression (Language: λ-expressions with subtyping)

Output:
- a well-typing for the input expression

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

We adopt the algorithm to Java

[3] [Fuh/Mishra 1988]: *Type inference with subtypes*, ESOP 1988

The algorithm consists of three sub-functions

- `tYPEClass`
- `match`
- `consistent` $\Rightarrow$ `typeUnify`

The sub-function **tYPE**

tYPE : $\texttt{TypeAssumptions} \times \texttt{class} \rightarrow \texttt{TypeAssumptions} \times \texttt{CoercionSet}$

- 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 sub-function **match**

$$\textbf{match} : \texttt{CoercionSet} \rightarrow \texttt{Substitution} \times \texttt{AtomicCoercionSet} + \{\, \mathit{fail} \,\}$$

- Reduce the coercions to atomic coercions (eliminate the function type constructors)

The sub-function **typeUnify**[3]

typeUnify:

$$\texttt{AtomicCoercionSet} \rightarrow (\{\,\texttt{Substitution}\,\}, \texttt{AtomicCoercionSet})$$

Input: $\{\,(\theta_1 \lessdot \theta'_1), \ldots, (\theta_1 \lessdot \theta'_1)\,\}$

Output: $(\{\,\sigma_1, \ldots, \sigma_m\,\}, AC')$

Post-condition:

$$\forall_i \{\,(\sigma_j(\,\theta_1\,) \leq^* \sigma_j(\,\theta'_1\,)), \ldots, (\sigma_j(\,\theta_1\,) \leq^* \sigma_j(\,\theta'_1\,))\,\}\ \&\&$$
$$AC' \subseteq \{\,(\theta_1 \lessdot \theta'_1), \ldots, (\theta_1 \lessdot \theta'_1)\,\},$$

where $\leq^*$ is the sub-typing relation

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

The whole algorithm

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

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

Implementation in Haskell

Overview

- Abstract syntax
- Function `tYPE`
- Function `match`
- Function `typeUnify`

Abstract syntax

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

data Stmt = ...

data StmtExpr = ...

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

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

tYPE function (with State-monad)

```
tYPEClass :: Class
          -> State (TypeAssumptions, Int) (CoercionSet, Class)
```

tYPE function (with State-monad)

```
tYPEClass :: Class
          -> State (TypeAssumptions, Int) (CoercionSet, Class)

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

tYPE function (with State-monad)

```
tYPEClass :: Class
          -> State (TypeAssumptions, Int) (CoercionSet, Class)

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

tYPEExprList :: [Expr]
             -> State (TypeAssumptions, Int) (CoercionSet, [Expr])
tYPEExprList (e : es) = (tYPEExpr e)
   >>= (\(v1, v1') -> (tYPEExprList es)
      >>= \(v2, v2') -> return (v1 ++ v2, v1' : v2'))
tYPEExprList [] = return ([], [])
```

tYPE function (with State-monad)

```
tYPEClass :: Class
          -> State (TypeAssumptions, Int) (CoercionSet, Class)

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

tYPEExprList :: [Expr]
             -> State (TypeAssumptions, Int) (CoercionSet, [Expr])
tYPEExprList (e : es) = (tYPEExpr e)
   >>= (\(v1, v1') -> (tYPEExprList es)
      >>= \(v2, v2') -> return (v1 ++ v2, v1' : v2'))
tYPEExprList [] = return ([], [])

tYPEExpr :: Expr ->
         -> State (TypeAssumptions, Int) (CoercionSet, Expr)
tYPEExpr e = ...
```

Example

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

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

}
```

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

match

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

type EquiTypes = [[Type]]

match :: CoercionSet -> CoercionSet
      -> State (Subst, EquiTypes) (Subst, CoercionSet)
```

match

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

type EquiTypes = [[Type]]

match :: CoercionSet -> CoercionSet
      -> State (Subst, EquiTypes) (Subst, CoercionSet)

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

match

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

type EquiTypes = [[Type]]

match :: CoercionSet -> CoercionSet
      -> State (Subst, EquiTypes) (Subst, CoercionSet)

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

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

match

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

type EquiTypes = [[Type]]

match :: CoercionSet -> CoercionSet
      -> State (Subst, EquiTypes) (Subst, CoercionSet)

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

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

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

match

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

type EquiTypes = [[Type]]

match :: CoercionSet -> CoercionSet
      -> State (Subst, EquiTypes) (Subst, CoercionSet)

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

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

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

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

Example

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

  V1 op =
   (# ((m:V2))
    -> (# ((f:V4))
    -> (((f:V4)).((Matrix.this:Matrix), (m:V2)):V8) :V3) :V1);

}
```

Example

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

  V1 op =
   (# ((m:V2))
    -> (# ((f:V4))
    -> (((f:V4)).((Matrix.this:Matrix), (m:V2)):V8) :V3) :V1);

}
```

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

typeUnify

```
typeUnify :: CoercionSet -> [Subst] × CoercionSet
```

Input: Set of (atomic) coercions

Output: List of substitutions and a subset of atomic Coercions, which are not unified

Implementation idea of typeUnify

`typeUnify :: CoercionSet -> [Subst]` $\times$ `CoercionSet`

Similar to the implementation of the Martelli/Montanari unification.
The rules:

- `reduce`
- `swap`
- `earse`
- `subst`

Implementation idea of typeUnify

`typeUnify :: CoercionSet -> [Subst]` $\times$ `CoercionSet`

Similar to the implementation of the Martelli/Montanari unification.
The rules:

- `reduce`
- `swap`
- `earse`
- `subst`

Additionally,

```
typeUnify ((a, Kl, ty') : coes) =
    [(ty,ty') : coes' | ty <- smaller ty', coes' <- typeUnify coes]

typeUnify ((ty, Kl, a) : coes) =
    [(ty,ty') : coes' | ty' <- greater ty, coes' <- typeUnify coes]
```

Example

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

   ##V13(#V21(V22, V23))(V10) op =
   (#{ ((m:V2))
    -> (#{ ((f:#V15(V16,V17)))
        -> (((f:#V15(V16,V17))).((Matrix.this:Matrix), (m:V2)):V8)}
        :#V11(#V18(V19,V20))) } : ##V13(#V21(V22, V23))(V10));

}
```

Example

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

   ##V13(#V21(V22, V23))(V10) op =
   (#{ ((m:V2))
    -> (#{ ((f:#V15(V16,V17)))
        -> (((f:#V15(V16,V17))).((Matrix.this:Matrix), (m:V2)):V8)}
        :#V11(#V18(V19,V20))) } : ##V13(#V21(V22, V23))(V10));

}
```

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

Typing of the Java program

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

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

Remainig coercions: $\texttt{V10} \leq^* \texttt{V2} \leq^* \texttt{V5} \leq^* \texttt{V17} \leq^* \texttt{V20} \leq^* \texttt{V23}$

Typing of the Java program

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

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

Remainig coercions: $\texttt{V10} \leq^* \texttt{V2} \leq^* \texttt{V5} \leq^* \texttt{V17} \leq^* \texttt{V20} \leq^* \texttt{V23}$

Equalizing: $\texttt{V10} = \texttt{V2} = \texttt{V5} = \texttt{V17} = \texttt{V20} = \texttt{V23}$

Typing of the Java program

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

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

Remainig coercions: $V10 \leq^* V2 \leq^* V5 \leq^* V17 \leq^* V20 \leq^* V23$

Equalizing: $V10 = V2 = V5 = V17 = V20 = V23$

Correct Java program

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

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

Principal type

```
##V13(#V21(Matrix,V10))(V10)
```

is **not** the principal type of op, as different instances would be possible.

Principal type

```
##V13(#V21(Matrix,V10))(V10)
```

is **not** the principal type of op, as different instances would be possible.

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

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

This declaration is not allowed in Java, as bounds must not be parameters.

Conclusion and Future work

Conclusion

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

Conclusion and Future work

Conclusion

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

Future work

- Extension of Java, such that bounded type variables are allowed as parameters
- IDE, such that the programmer can select a type
- Introducing of intersection-types for Java–functions, such that all inferred types are usable
- Overloading