Well-typings for Java$_\lambda$

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

4. Mai 2011

Overview

History of Java type system

Version 1:

- Subtyping on classes (without parameters)
  Ex.: $\texttt{Integer} \leq^* \texttt{Object}$

History of Java type system

Version 1:

- Subtyping on classes (without parameters)
  Ex.: `Integer` $\leq^*$ `Object`

Version 5:

- Parametrized classes (type constructors)
  Ex.: `Vector<X>`
- Subtyping extension
  Ex.: `Matrix` $\leq^*$ `Vector<Vector<Integer>>`
- Wildcards (with lower and upper bounds)
  Ex.: `Vector<? extends Integer>`

History of Java type system

Version 1:

- Subtyping on classes (without parameters)
  Ex.: `Integer` $\leq^*$ `Object`

Version 5:

- Parametrized classes (type constructors)
  Ex.: `Vector<X>`
- Subtyping extension
  Ex.: `Matrix` $\leq^*$ `Vector<Vector<Integer>>`
- Wildcards (with lower and upper bounds)
  Ex.: `Vector<? extends Integer>`

Version 8:

- Closures (λ–expressions)
- SAM–Types, but no function types

Single Abstract Method–Types

- abstract class with a single abstract method or
- interface with a single method declaration

Example:

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

call by an anonymous inner class:

```
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 })
```

Function type declaration

In earlier announcments (e.g. Project Lambda[1] Java Language Specification draft (Version 0.1.5)) explicite function types had been introduced:

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

More convienent declaration:

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

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

Function type declaration

In earlier announcments (e.g. Project Lambda[1] Java Language Specification draft (Version 0.1.5)) explicite function types had been introduced:

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

More convienent declaration:

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

Function types in Java have some disadvantages (Brian Goetz)

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

Function type declaration

In earlier announcments (e.g. Project Lambda[1] Java Language Specification draft (Version 0.1.5)) explicite function types had been introduced:

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

More convienent declaration:

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

Function types in Java have some disadvantages (Brian Goetz)

Type inference could solve some problems (Martin Plümicke)

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

The language

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

Adapt Fuh and Mishra's algorithm

- Java$_\lambda$ type system is equivalent
- subtyping, but
- no overloading

Adapt Fuh and Mishra's algorithm

- Java$_\lambda$ type system is equivalent
- subtyping, but
- no overloading
- Fuh and Mishra's algorithm determines *well typings*

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

  - C = set of coercions
  - A = set of type assumptions
  - N = Ausdruck
  - t = Typ

The algorithm I

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

The algorithm I

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

TYPE : $\texttt{TypeAssumptions} \times \texttt{class} \longrightarrow \texttt{TypeAssumptions} \times \texttt{CoercionSet}$

Maps a fresh type variable to each subterm of the functions and determines a result type (variable) for each function and the corresponding coercions

MATCH : $\texttt{CoercionSet} \longrightarrow \texttt{Substitution} \times \texttt{ACoercionSet} + \{\, \textit{fail} \,\}$

Extended type unification to adopt the structur of the coercions. Transform to atomic coercions (eliminate type constructors)

CONSISTENT : $\texttt{AtomicCoercionSet} \longrightarrow \texttt{Boolean}$

Consistence check of the atomic coercions.

The algorithm II

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

WTYPE$(\, \mathit{Ass}, \mathsf{Class}(\, \mathit{cl}, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}, \mathit{ivardecls} \,) \,) =$

let

$(\{\, f_1 : a_1, \ldots, f_n : a_n \,\}, \mathit{CoeS}) =$

TYPE$(\, \mathit{Ass}, \mathsf{Class}(\, \mathit{cl}, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}, \mathit{ivardecls} \,) \,)$

$(\sigma, \mathit{AC}) =$ **MATCH**$(\, \mathit{CoeS} \,)$

in

if CONSISTENT$(\, \mathit{AC} \,)$ **then**

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

else *fail*

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) } }

  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 ();
                        : //matrice multiplication
                        return ret;
                    })
  }
}
```

TYPE

```
a_op op =
  a_#m #{ a_m m ->
          a_#f #{ a_f f ->
                  a_f( M.this,m ) f(Matrix Matrix.this, a_m m) } }
```

TYPE

```
a_op op =
  a_#m #{ a_m m ->
          a_#f #{ a_f f ->
                  a_f(M.this,m) f(Matrix Matrix.this, a_m m) } }
```

- $a_{\#\mathtt{m}} \lessdot a_{op}$
- $a_m \longrightarrow a_{\#f} \lessdot a_{\#\mathtt{m}}$
- $a_f \longrightarrow a_{f(\,this,m\,)} \lessdot a_{\#f}$
- $a_f \lessdot (a_1, a_2) \longrightarrow a_3$
- $\mathtt{Matrix} \lessdot a_1$
- $a_m \lessdot a_2$
- $a_3 \lessdot a_{f(\,this,m\,)}$

MATCH

Matched coercions:

- $\beta \to ((\epsilon_3, \epsilon_3') \to \epsilon_3'') \to \gamma_2' \lessdot \beta_1 \to ((\epsilon_4, \epsilon_4') \to \epsilon_4'') \to \gamma_3'$,
- $a_m \to ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1' \lessdot \beta \to ((\epsilon_3, \epsilon_3') \to \epsilon_3'') \to \gamma_2'$,
- $((\epsilon_1, \epsilon_1') \to \epsilon_1'') \to a_{f(\,this,m\,)} \lessdot ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1'$
- $(\epsilon_1, \epsilon_1') \to \epsilon_1'' \lessdot (a_1, a_2) \to a_3$
- $\texttt{Matrix} \lessdot a_1$
- $a_m \lessdot a_2$
- $a_3 \lessdot a_{f(\,this,m\,)}$

MATCH

Matched coercions:

- $\beta \to ((\epsilon_3, \epsilon'_3) \to \epsilon''_3) \to \gamma'_2 \lessdot \beta_1 \to ((\epsilon_4, \epsilon'_4) \to \epsilon''_4) \to \gamma'_3$,
- $a_m \to ((\epsilon_2, \epsilon'_2) \to \epsilon''_2) \to \gamma'_1 \lessdot \beta \to ((\epsilon_3, \epsilon'_3) \to \epsilon''_3) \to \gamma'_2$,
- $((\epsilon_1, \epsilon'_1) \to \epsilon''_1) \to a_{f(\,this,m\,)} \lessdot ((\epsilon_2, \epsilon'_2) \to \epsilon''_2) \to \gamma'_1$
- $(\epsilon_1, \epsilon'_1) \to \epsilon''_1 \lessdot (a_1, a_2) \to a_3$
- $\texttt{Matrix} \lessdot a_1$
- $a_m \lessdot a_2$
- $a_3 \lessdot a_{f(\,this,m\,)}$

Atomic coercions: $AC =$

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

CONSISTENCE

It	*Coercion*	$l_{\mathtt{M}}$	l_{a_1}	l_{ϵ_1}	l_{ϵ_2}	l_{ϵ_3}	l_{ϵ_4}
0		M	$*$	$*$	$*$	$*$	$*$
1	$\mathtt{M} \lessdot a_1$	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>>

CONSISTENCE

It	*Coercion*	l_M	l_{a_1}	l_{ϵ_1}	l_{ϵ_2}	l_{ϵ_3}	l_{ϵ_4}
0		M	$*$	$*$	$*$	$*$	$*$
1	$M \lessdot a_1$	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>>

Result:

$$(AC, Ass_1) \vdash \{\, \mathrm{op} : \beta_1 \to ((\epsilon_4, \epsilon_4') \to \epsilon_4'') \to \gamma_3' \,\}$$

with

$\epsilon_4 = \mathtt{Matrix, Vector<Vector<Integer>>}$

$\beta_1 \lessdot a_m \lessdot a_2 \lessdot \epsilon_1' \lessdot \epsilon_2' \lessdot \epsilon_3' \lessdot \epsilon_4'$

$\epsilon_4'' \lessdot \epsilon_4'' \lessdot \epsilon_2'' \lessdot \epsilon_1'' \lessdot a_3 \lessdot a_{f(\,this, m\,)} \lessdot \gamma_1' \lessdot \gamma_2' \lessdot \gamma_3'$

Different Solutions (Intersection Type)

$$\mathrm{op} : \beta_1 \longrightarrow ((\epsilon_4, \epsilon_4') \longrightarrow \epsilon_4'') \longrightarrow \gamma_3'$$

with

a_1	ϵ_1	ϵ_2	ϵ_3	ϵ_4	β_1	ϵ_4''	...
M	M	M	M	**M**	ϵ_4'	γ_3'	
M	M	M	M	**V<V<Int>>**	ϵ_4'	γ_3'	
M	M	M	V<V<Int>>	V<V<Int>>	ϵ_4'	γ_3'	
M	M	V<V<Int>>	V<V<Int>>	V<V<Int>>	ϵ_4'	γ_3'	
M	V<V<Int>>	V<V<Int>>	V<V<Int>>	V<V<Int>>	ϵ_4'	γ_3'	
V<V<Int>>	V<V<Int>>	V<V<Int>>	V<V<Int>>	V<V<Int>>	ϵ_4'	γ_3'	

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) }

}
```

Related work

Scala:

- closures, function-types
- pattern-matching and currying
- local type inference

  no type inference for whole λ–expressions and recursive methods

Related work

Scala:

- closures, function-types
- pattern-matching and currying
- local type inference

  no type inference for whole λ–expressions and recursive methods

C#:

- closures
- function types as delegates (similar to SAM-types)
- no type inference

Conclusion and Future work

Conclusion

- Fuh and Mishra's type inference algorithm can be adopted to Java$_\lambda$.
- *Real function types* are possible without confusing the programmers
- *Well typings* are results of the type inference algorithm
  - Intersection type approach
  - Bounded type variables approach

Conclusion and Future work

Conclusion

- Fuh and Mishra's type inference algorithm can be adopted to Java$_\lambda$.
- *Real function types* are possible without confusing the programmers
- *Well typings* are results of the type inference algorithm
  - Intersection type approach
  - Bounded type variables approach

Future work

- Implementation of the intersection approach by IDE
- Extension of the byte code for the realisiation of coercions as bounded type variables
- Overloading