Well-typings for Java$_\lambda$

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

August, 25th 2011

Overview

History of Java type system

Version 1:

- Subtyping on classes (without parameters)
  Ex.: `Integer` $\leq^*$ `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. Mark Reinhold: Project Lambda[1] Java Language Specification draft (Version 0.1.5)) explicite function types had been introduced:

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

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

Function type declaration

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

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

Different approaches:

Function types

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

SAM-Types

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

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

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

The language

Source	:=	*class*∗
class	:=	Class(*stype*, [extends(*stype*),]*IVarDecl*∗, *FunDecl*∗)
IVarDecl	:=	InstVarDecl(*stype*, *var*)
FunDecl	:=	Fun(*fname*, [*type*], *lambdaexpr*)
block	:=	Block(*stmt*∗)
stmt	:=	*block* | Return(*expr*) | While(*bexpr*, *block*)
	|	LocalVarDecl(*var*[, *type*]) | If(*bexpr*, *block*[, *block*])
	|	*stmtexpr*
lambdaexpr	:=	Lambda(((*var*[, *type*]))∗, (*stmt* | *expr*))
stmtexpr	:=	Assign(*var*, *expr*) | New(*stype*, *expr*∗)
	|	Eval(*expr*, *expr*∗)
expr	:=	*lambdaexpr* | *stmtexpr* | this | This(*stype*) | super
	|	LocalOrFieldVar(*var*) | InstVar(*expr*, *var*)
	|	InstFun(*expr*, *fname*) | *bexp* | *sexp*

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) } }

  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) } }


  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;
                  })
  }
}
```

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

Java has no well-typings!!!

The algorithm I

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

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

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

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

- 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 $: \texttt{CoercionSet} \rightarrow \texttt{Substitution} \times \texttt{ACoercionSet} + \{\, \textit{fail} \,\}$

- Type unification [Pluemicke 2009][2] to adapt the structure of the coercions.
- Reduce the coercions to atomic coercions (eliminate type constructors)

[2] [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

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

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

let

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

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

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

in

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

  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;
                    })
  }
}
```

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/Unified 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/Unified coercions:

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

Reduced 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*	$I_{\mathtt{M}}$	I_{a_1}	$I_{\epsilon 1}$	$I_{\epsilon 2}$	$I_{\epsilon 3}$	$I_{\epsilon 4}$
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	$I_{\mathtt{M}}$	I_{a_1}	I_{ϵ_1}	I_{ϵ_2}	I_{ϵ_3}	I_{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>>

Result:

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

CONSISTENCE

It	$Coercion$	$I_{\mathtt{M}}$	I_{a_1}	$I_{\epsilon 1}$	$I_{\epsilon 2}$	$I_{\epsilon 3}$	$I_{\epsilon 4}$
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>>

Result:

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

Comparison: Declared type:

$$\mathrm{op} : \mathtt{Matrix} \to ((\mathtt{Matrix},\mathtt{Matrix}) \to \mathtt{Matrix}) \to \mathtt{Matrix}$$

The well-typing is correct but not very meaningful!

Different Solutions (Intersection Type)

$$\texttt{op} : \beta_1 \rightarrow ((\epsilon_4, \epsilon'_4) \rightarrow \epsilon''_4) \rightarrow \gamma'_3$$

A modification of the algorithm CONSISTENCE results:

- $\epsilon_4 = \texttt{Matrix}, \texttt{Vector<Vector<Integer>>}$
- $\beta_1 \lessdot a_m \lessdot a_2 \lessdot \epsilon'_1 \lessdot \epsilon'_2 \lessdot \epsilon'_3 \lessdot \epsilon'_4$
- $\epsilon''_4 \lessdot \epsilon''_4 \lessdot \epsilon''_2 \lessdot \epsilon''_1 \lessdot a_3 \lessdot a_{f(\,this,m\,)} \lessdot \gamma'_1 \lessdot \gamma'_2 \lessdot \gamma'_3$

Different Solutions (Intersection Type)

$$\texttt{op} : \beta_1 \rightarrow ((\epsilon_4, \epsilon_4') \rightarrow \epsilon_4'') \rightarrow \gamma_3'$$

A modification of the algorithm CONSISTENCE results:

- $\epsilon_4 = \texttt{Matrix}, \texttt{Vector<Vector<Integer>>}$
- $\beta_1 \lessdot a_m \lessdot a_2 \lessdot \epsilon_1' \lessdot \epsilon_2' \lessdot \epsilon_3' \lessdot \epsilon_4'$
- $\epsilon_4'' \lessdot \epsilon_4'' \lessdot \epsilon_2'' \lessdot \epsilon_1'' \lessdot a_3 \lessdot a_{f(\,this,m\,)} \lessdot \gamma_1' \lessdot \gamma_2' \lessdot \gamma_3'$

This result is meaningful and more principal than the declared type.

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) }

}
```

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 as principal as the inferred type.

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 as principal as the inferred type.
But not allowed in Java.

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)
- type inference for `var` declarations

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 type declarations
- *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_λ.
- *Real function types* are possible without confusing type declarations
- *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