Formalization of the Java$_\lambda$ type system

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

3. Mai 2010

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 7:

- ??? Closures (λ–expressions) ???

Java 7: Bulk-data APIs like parallel arrays

- parallelism approach: sorting, searching, selection

Example:

```
public class Student {
    String name;
    int graduationYear;
    double gpa;
}

ParallelArray<Student> students
    = new ParallelArray<Student>(fjPool, data);

double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();
```

Realization by helper objects

```
static final Ops.Predicate<Student> isSenior
    = new Ops.Predicate<Student>() {
        public boolean op(Student s) {
            return s.graduationYear == Student.THIS_YEAR;
        }
    };

static final Ops.ObjectToDouble<Student> selectGpa
    = new Ops.ObjectToDouble<Student>() {
        public double op(Student student) {
            return student.gpa;
        }
    };
```

Realization by closures (λ–expressions)

```
double bestGpa
    = students.withFilter(
                  #(Student s)(s.graduationYear == THIS_YEAR))
              .withMapping(#(Student s)(s.gpa))
              .max();
```

Different approaches

- Closures for the Java Programming Language: BGGA
  [Bracha, Gafter, Gosling, von der Ahé]
- Concise Instance Creation Expressions: Closures without Complexity: CICE
  [Lee, Lea, Bloch]
- First-class methods: Java-style closures: FCM
  [Colebourne, Schulz]

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

Different approaches

- Closures for the Java Programming Language: BGGA
  [Bracha, Gafter, Gosling, von der Ahé]
- Concise Instance Creation Expressions: Closures without Complexity: CICE
  [Lee, Lea, Bloch]
- First-class methods: Java-style closures: FCM
  [Colebourne, Schulz]

Our approach is following:
Project Lambda[1] Java Language Specification draft (Version 0.1.5)

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

Mark Reinhold's Blog (Principal Engineer Java SE and OpenJDK)

Two key features are needed:

- A literal syntax, for writing closures, and
- Function types, so that closures are first-class citizens in the type system.

To integrate closures with the rest of the language and the platform we need two additional features:

- Closure conversion to implement a single-method interface or abstract class and
- Extension methods

The language

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

Simple types $\mathrm{SType}_{TS}(\,BTV\,)$

- $BTV^{(ty)} \subseteq \mathrm{SType}_{TS}(\,BTV\,)$ (bounded type variables)
- $TC^{()} \subseteq \mathrm{SType}_{TS}(\,BTV\,)$ (0-ary type constructors/classes)
- For $ty_i \in \mathrm{SType}_{TS}(\,BTV\,)$
  $\cup \{\,\texttt{?}\,\}$
  $\cup \{\,\texttt{? extends}\ \tau \mid \tau \in \mathrm{SType}_{TS}(\,BTV\,)\,\}$
  $\cup \{\,\texttt{? super}\ \tau \mid \tau \in \mathrm{SType}_{TS}(\,BTV\,)\,\}$

  and $C \in TC^{(a_1|_{b_1} \ldots a_n|_{b_n})}$ it holds

$$C\texttt{<}ty_1, \ldots, ty_n\texttt{>} \in \mathrm{SType}_{TS}(\,BTV\,)$$

if for $CC(\,C\texttt{<}ty_1, \ldots, ty_n\texttt{>}\,) = C\texttt{<}\overline{ty_1}, \ldots, \overline{ty_n}\texttt{>}$ holds:

$$\overline{ty_i} \leq^* b_i[a_j \mapsto \overline{ty_j} \mid 1 \leqslant j \leqslant n],$$

where

- $CC(\ldots)$ denotes the capture conversion
- $\leq^*$ is the subtyping ordering.

Types $\mathsf{Type}_{TS}(\ BTV\ )$

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

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

Subtyping–relation

Let $\leq^*$ the Java subtyping relation on simple types $\mathrm{SType}_{TS}(\,BTV\,)$.

The continuation on $\mathrm{Type}_{TS}(\,BTV\,)$ is defined as:

$$\#\,\theta_0\,(\theta'_1,\ldots,\theta'_n) \leq^* \#\,\theta'_0\,(\theta_1,\ldots,\theta_n) \quad \textit{iff} \quad \theta_i \leq^* \theta'_i.$$

λ–Abstraction

$$[\textbf{lambda}_{\textbf{stmt}}] \quad \frac{(O \cup \{ x_i : \theta_i \}, \tau, \tau') \rhd_{Stmt} \; s : \theta}{(O, \tau, \tau') \rhd_{Expr} \; \mathsf{Lambda}(\,(x_1, \ldots, x_n), s\,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

λ–Abstraction

$$[\textbf{lambda}_{\textbf{stmt}}] \quad \frac{(O \cup \{ x_i : \theta_i \}, \tau, \tau') \rhd_{Stmt} \; s : \theta}{(O, \tau, \tau') \rhd_{Expr} \; \mathsf{Lambda}(\,(x_1, \ldots, x_n), s\,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

$$[\textbf{lambda}_{\textbf{expr}}] \quad \frac{(O \cup \{ x_i : \theta_i \}, \tau, \tau') \rhd_{Expr} \; e : \theta}{(O, \tau, \tau') \rhd_{Expr} \; \mathsf{Lambda}(\,(x_1, \ldots, x_n), e\,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

Application

[App]

$$\frac{(O, \tau, \tau') \rhd_{Expr} e : \#\,\theta\,(\theta'_1, \ldots, \theta'_n), \quad (O, \tau, \tau') \rhd_{Expr} e_i : \theta_i}{(O, \tau, \tau') \rhd_{Expr} \mathsf{Eval}(\, e, e_1 \ldots e_n\,) : \theta} \; \theta_i \leq^* \theta'_i$$

Application

[App]

$$\frac{(O,\tau,\tau') \rhd_{Expr} e : \#\,\theta\,(\theta'_1,\ldots,\theta'_n), \quad (O,\tau,\tau') \rhd_{Expr} e_i : \theta_i}{(O,\tau,\tau') \rhd_{Expr} \mathsf{Eval}(\, e, e_1 \ldots e_n \,) : \theta} \; \theta_i \leq^* \theta'_i$$

[InstFun]

$$\frac{(O,\tau,\tau') \rhd_{Expr} re : \overline{\theta}, \qquad O_{\overline{\theta}} \rhd_{Id} f : \#\,\theta\,(\theta_1,\ldots,\theta_n)}{(O,\tau,\tau') \rhd_{Expr} \mathsf{InstFun}(\, re, f \,) : \#\,\theta\,(\theta_1,\ldots,\theta_n)}$$

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
- **Problem**: Fuh and Mishra's algorithm determines *well typings* (set of possibly not unique solvable unequations)

Fuh und Mishra's algorithm

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

[2] [Martelli, Montanari 1982]: *An Efficient Unification Algorithm*

Fuh und Mishra's algorithm

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

MATCH : $\texttt{CoercionSet} \rightarrow \texttt{Substitution} + \{ \mathit{fail} \}$

Extended unification to adopt the structur of the coercions.
Derived from [Martelli, Montanari 1982][2]

[2] [Martelli, Montanari 1982]: *An Efficient Unification Algorithm*

Fuh und Mishra's algorithm

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

MATCH : $\texttt{CoercionSet} \rightarrow \texttt{Substitution} + \{ \mathit{fail} \}$

Extended unification to adopt the structur of the coercions. Derived from [Martelli, Montanari 1982][2]

SIMPLIFY : $\texttt{CoercionSet} \rightarrow \texttt{AtomicCoercionSet}$

Eliminate type constructors, especially the *functor* and the *tuple-construction*

[2] [Martelli, Montanari 1982]: *An Efficient Unification Algorithm*

Fuh und Mishra's algorithm

WTYPE : $\texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{WellTyping} + \{ \textit{fail} \}$

MATCH : $\texttt{CoercionSet} \rightarrow \texttt{Substitution} + \{ \textit{fail} \}$

Extended unification to adopt the structur of the coercions. Derived from [Martelli, Montanari 1982][2]

SIMPLIFY : $\texttt{CoercionSet} \rightarrow \texttt{AtomicCoercionSet}$

Eliminate type constructors, especially the *functor* and the *tuple-construction*

CONSISTENT : $\texttt{AtomicCoercionSet} \rightarrow \texttt{Boolean} + \{ \textit{fail} \}$

Consistence check, and additionally determination of possible solutions.

[2] [Martelli, Montanari 1982]: *An Efficient Unification Algorithm*

Adaption to the Java$_\lambda$ type system

SIMPLIFY: The [Martelli, Montanari 1982] unification is substituted by the [Pluemicke 2009] type unification[3]

CONSISTENCE: The functions *above* and *below* are substituted by the functions `greater` and `smaller`[3], as *above* and *below* are not finite in the Java$_\lambda$ type system.

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

Example

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

  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
    op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(this, m)))
  }
```

Example

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

  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
    op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(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;
                      })
  }
}
```

Goal

Typed syntax:

```
##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(this, m)))
```

Goal

Typed syntax:

```
##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(this, m)))
```

Typeless syntax:

```
  op = #(m)(#(f)(f(this, m)))
```

Goal: The system determines the type

```
##Matrix(#Matrix(Matrix, Matrix))(Matrix)
```

automatically.

$WTYPE(\emptyset, \#(\mathtt{m})(\#(\mathtt{f})(\mathtt{f}.(\mathtt{this}, \mathtt{m})))),$ (TYPE)

- $t_m \longrightarrow t_{\#f} \lessdot t_{\mathrm{op}}$
- $t_f \longrightarrow t_{f(\,this,m\,)} \lessdot t_{\#f}$
- $t_f \lessdot (t_1, t_2) \longrightarrow t_3$
- $\mathtt{Matrix} \lessdot t_1$
- $t_m \lessdot t_2$
- $t_3 \lessdot t_{f(\,this,m\,)}$

$WTYPE(\emptyset, \#(\mathtt{m})(\#(\mathtt{f})(\mathtt{f}.(\mathtt{this}, \mathtt{m}))))$, (MATCH)

- $t_m \longrightarrow ((\epsilon_2, \epsilon_2') \longrightarrow \epsilon_2'') \longrightarrow \gamma_1' \lessdot \beta \longrightarrow ((\epsilon_3, \epsilon_3') \longrightarrow \epsilon_3'') \longrightarrow \gamma_2'$,
  $(t_{\mathsf{op}} \mapsto \beta \longrightarrow \beta')$, $(t_{\#f} \mapsto \gamma_1 \longrightarrow \gamma_1')$, $(\beta' \mapsto \gamma_2 \longrightarrow \gamma_2')$,
  $(\gamma_1 \mapsto (\epsilon_2, \epsilon_2') \longrightarrow \epsilon_2'')$, $(\gamma_2 \mapsto (\epsilon_3, \epsilon_3') \longrightarrow \epsilon_3'')$
- $((\epsilon_1, \epsilon_1') \longrightarrow \epsilon_1'') \longrightarrow t_{f(\,this,m\,)} \lessdot ((\epsilon_2, \epsilon_2') \longrightarrow \epsilon_2'') \longrightarrow \gamma_1'$
  $(t_{\#f} \mapsto \gamma_1 \longrightarrow \gamma_1')$, $(t_f \mapsto (\epsilon_1, \epsilon_1') \longrightarrow \epsilon_1'')$, $(\gamma_1 \mapsto (\epsilon_2, \epsilon_2') \longrightarrow \epsilon_2'')$
- $(\epsilon_1, \epsilon_1') \longrightarrow \epsilon_1'' \lessdot (t_1, t_2) \longrightarrow t_3$
  $(t_f \mapsto (\epsilon_1, \epsilon_1') \longrightarrow \epsilon_1'')$, $(\gamma_1 \mapsto (\epsilon_2, \epsilon_2') \longrightarrow \epsilon_2'')$
- $\mathtt{Matrix} \lessdot t_1$
- $t_m \lessdot t_2$
- $t_3 \lessdot t_{f(\,this,m\,)}$

$WTYPE(\emptyset, \#(\mathtt{m})(\#(\mathtt{f})(\mathtt{f}.(\mathtt{this}, \mathtt{m}))))$, (SIMPLIFY)

- $\beta \lessdot t_m$,
  $\epsilon_2 \lessdot \epsilon_3$, $\epsilon'_2 \lessdot \epsilon'_3$, $\epsilon''_3 \lessdot \epsilon''_2$,
  $\gamma'_1 \lessdot \gamma'_2$
- $\epsilon_1 \lessdot \epsilon_2$, $\epsilon'_1 \lessdot \epsilon'_2$, $\epsilon''_2 \lessdot \epsilon''_1$,
  $t_{f(\,this,m\,)} \lessdot \gamma'_1$
- $t_1 \lessdot \epsilon_1$,
  $t_2 \lessdot \epsilon'_1$,
  $\epsilon''_1 \lessdot t_3$
- $\mathtt{Matrix} \lessdot t_1$
- $t_m \lessdot t_2$
- $t_3 \lessdot t_{f(\,this,m\,)}$

$WTYPE(\emptyset, \#(\mathtt{m})(\#(\mathtt{f})(\mathtt{f}.(\mathtt{this}, \mathtt{m}))))$, (CONSISTENCE)

It	*Coercion*	$I_{\mathtt{Matrix}}$	I_{t_1}	I_{ϵ_1}	I_{ϵ_2}	I_{ϵ_3}	...
0		M	*	*	*	*	
1	$\mathtt{M} \lessdot t_1$	M	M, V<V<Int>>	*	*	*	
1	$t_1 \lessdot \epsilon_1$	M	M, V<V<Int>>	M, V<V<Int>>	*	*	
1	$\epsilon_1 \lessdot \epsilon_2$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	*	
1	$\epsilon_2 \lessdot \epsilon_3$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	
1	...	M	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>>	

$WTYPE(\emptyset, \#(\mathtt{m})(\#(\mathtt{f})(\mathtt{f}.(\mathtt{this}, \mathtt{m}))))$, (CONSISTENCE)

It	$Coercion$	$I_{\mathtt{Matrix}}$	I_{t_1}	I_{ϵ_1}	I_{ϵ_2}	I_{ϵ_3}	...
0		M	*	*	*	*	
1	$\mathtt{M} \lessdot t_1$	M	M, V<V<Int>>	*	*	*	
1	$t_1 \lessdot \epsilon_1$	M	M, V<V<Int>>	M, V<V<Int>>	*	*	
1	$\epsilon_1 \lessdot \epsilon_2$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	*	
1	$\epsilon_2 \lessdot \epsilon_3$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	
1	...	M	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>>	

Result:

$$\mathrm{op} : \beta \to ((\epsilon_3, \epsilon_3') \to \epsilon_3'') \to \gamma_2'$$

with

$\epsilon_3 = \mathtt{Matrix}, \mathtt{Vector<Vector<Integer>>}$

$\beta \lessdot t_m \lessdot t_2 \lessdot \epsilon_1' \lessdot \epsilon_2' \lessdot \epsilon_3'$

$\epsilon_3'' \lessdot \epsilon_2'' \lessdot \epsilon_1'' \lessdot t_3 \lessdot t_{f(\,this, m\,)} \lessdot \gamma_1' \lessdot \gamma_2'$

Conclusion and Future work

Conclusion

- Closures in Java: Key Features are *syntax for writing closures* and *function types as first-class citizens*.
- Java type system is equivalent to the type system of the Fuh and Mishra's type inference algorithm
- *Well typings* are results of the Fuh and Mishra's type inference algorithm

Conclusion and Future work

Conclusion

- Closures in Java: Key Features are *syntax for writing closures* and *function types as first-class citizens*.
- Java type system is equivalent to the type system of the Fuh and Mishra's type inference algorithm
- *Well typings* are results of the Fuh and Mishra's type inference algorithm

Future work

- Open question: How to integrate *well typings* in the Java's type system?
- Correctness proof of the adapted type inference algorithm
- Implementation