Intersection Types in Java

Martin Plümicke

University of Cooperative Education
Stuttgart/Horb

September 11, 2008
Overview

Motivation/Tooldemonstration

Method calls with intersection types
  Semantics
  First approach
  The algorithm

Principal Typing

Conclusion and Outlook
Purpuse:
Byte-code generation for methods with intersection types

Intersection Types in Java
**Purpose:** Byte-code generation for methods with intersection types

Martin Plümicke  
Intersection Types in Java
Semantics of type–inferred Java programs

- control-structures have the same semantics as in standard Java
- method calls differ, as there are intersection types
control-structures have the same semantics as in standard Java

method calls differ, as there are intersection types

```
... m ( ... ) {
    ... receiver.method(t1, ..., tn); ... 
}
```

method: \( ty_1 \times \ldots \times ty_n \rightarrow ty_0 \) & \( \ldots \) & \( ty'_1 \times \ldots \times ty'_n \rightarrow ty'_0 \)

has an intersection type
Semantics of type–inferred Java programs

- control-structures have the same semantics as in standard Java
- method calls differ, as there are intersection types

```java
... m ( ... ) {
  ... receiver.method(t1, ..., tn); ...
}
```

- method: $t_1 \times \ldots \times t_n \rightarrow t_0$ & $\ldots$ & $t'_1 \times \ldots \times t'_n \rightarrow t'_0$
  - has an intersection type

- $t_1, \ldots, t_n$ have unambiguous types during execution
- by the argument types and the result type the typing of method is determined

- method is executed with the determined typing.
Semantics example

class OL {
    Integer m(x) { return x + x; }    //Integer → Integer
    Boolean m(x) { return x || x; }   //Boolean → Boolean
}

class Main {
    main(x) {       // Integer → Integer & Boolean → Boolean
        ol;
        ol = new OL();
        return ol.m(x);
    }
}

...  
Main rec = new Main();  
Integer r = rec.main(x);
class OL {
    Integer m(x) { return x + x; }   //Integer → Integer
    Boolean m(x) { return x || x; }   //Boolean → Boolean
}

class Main {
    main(x) { // Integer → Integer & Boolean → Boolean
        ol;
        ol = new OL();
        return ol.m(x);
    }
}

...  
Main rec = new Main();
Integer r = rec.main(x); main:Integer → Integer is determined
Code generation for method with intersection types

- Byte-code allows no intersection types
- First approach: generate for each element of the intersection type an own method

```java
class Main {
    Integer main(Integer x) {
        OL ol = new OL();
        return ol.m(x);
    }

    Boolean main(Boolean x) {
        OL ol = new OL();
        return ol.m(x);
    }
}
```
Byte-code allows no intersection types
First approach: generate for each element of the intersection type an own method

Result for Main:

```java
class Main {
    Integer main(Integer x) {
        OL ol;
        ol = new OL();
        return ol.m(x); }

    Boolean main(Boolean x) {
        OL ol;
        ol = new OL();
        return ol.m(x); }
}
```
Example: Multiplication of matrices I

class Matrix extends Vector<Vector<Integer>> {
    mul(m) {
        ret = new Matrix();
        int i = 0;
        while(i < size()) {
            v1; v1 = this.elementAt(i);
            v2; v2 = new Vector<Integer>();
            int j = 0;
            while(j < v1.size()) {
                int erg = 0;
                int k = 0;
                while(k < v1.size()) {
                    erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j); k++;
                }
                v2.addElement(new Integer(erg)); j++;
            }
            ret.addElement(v2); i++;
        }
        return ret;
    }
}
Example: Multiplication of matrices II

\[
\text{mul} : \&_{\beta,\alpha}(\beta \rightarrow \alpha),
\]

where

\[
\beta \leq^{*} \text{Vector}\langle ? \text{extends Vector}\langle ? \text{extends Integer} \rangle \rangle, \quad \text{Matrix} \leq^{*} \alpha
\]
Example: Multiplication of matrices II

\[
\text{mul} : &_{\beta,\alpha}(\beta \rightarrow \alpha),
\]

where

\[
\beta \leq^* \text{Vector}\langle ? \text{ extends Vector}\langle ? \text{ extends Integer}\rangle\rangle,
\]

Matrix \leq^* \alpha

class Matrix extends Vector<Vector<Integer>> {
    Matrix mul(Vector<? extends Vector<? extends Integer>> m) {
        ...;
    }
    Matrix mul(Vector<? extends Vector<Integer>> m) {
        ...;
    }
    Matrix mul(Vector<Vector<Integer>> m) {
        ...;
    }
    ...
    Vector<Vector<Integer>> mul(Vector<Vector<Integer>> m) {
        ...;
    }
    ...
    Vector<? extends Vector<? extends Integer>> mul(Matrix m) {
        ...;
    }
}
Example: Multiplication of matrices II

\[ \text{mul: } \&_{\beta, \alpha}(\beta \rightarrow \alpha) , \]

where

\[ \beta \leq^* \text{Vector}\langle? \text{extends Vector}\langle? \text{extends Integer}\rangle\rangle, \]
\[ \text{Matrix} \leq^* \alpha \]

\begin{verbatim}
class Matrix extends Vector<Vector<Integer>> { 
    Matrix mul(Vector<? extends Vector<? extends Integer>> m) { ... }
    Matrix mul(Vector<? extends Vector<Integer>> m) { ... }
    Matrix mul(Vector<Vector<Integer>> m) { ... }
    ... 
    Vector<Vector<Integer>> mul(Vector<Vector<Integer>> m) { ... }
    ... 
    Vector<? extends Vector<? extends Integer>> mul(Matrix m) { ... }
}
\end{verbatim}

Not a correct Java program
Group elements of the intersection type

Idea:

1. Group all elements which
   ▶ executes the same code
   ▶ have a common supertype
2. Generate new methods only for the groups
Group elements of the intersection type

Idea:
1. Group all elements which
   ▶ executes the same code
   ▶ have a common supertype
2. Generate new methods only for the groups

Code–execution: Callgraph of the method declarations

\[ CG(\ cl.m : \tau ) \]

Callgraph of the method \( m \) in the class \( cl \) with the typing \( \tau \).
Group elements of the intersection type

Idea:
1. Group all elements which
   ▶ executes the same code
   ▶ have a common supertype
2. Generate new methods only for the groups

Code–execution: Callgraph of the method declarations

\[ CG(\text{cl}.m : \tau) \]

Callgraph of the method \( m \) in the class \( \text{cl} \) with the typing \( \tau \).

Supertype of function types: Subtyping ordering
\[
\theta_i \leq^* \theta'_i, \theta \leq^* \theta' \Rightarrow \\
\theta_1 \times \ldots \times \theta_n \rightarrow \theta' \leq^* \theta'_1 \times \ldots \times \theta'_n \rightarrow \theta
\]
Example class OL 1

Callgraph

\[ CG(\text{Main.main: Integer} \rightarrow \text{Integer}) \quad CG(\text{Main.main: Boolean} \rightarrow \text{Boolean}) \]

\[ = \]

\[ \text{Main.main: Integer} \rightarrow \text{Integer} \]
\[ & \text{Boolean} \rightarrow \text{Boolean} \]

\[ \Downarrow \]

\[ \text{OL.m: Integer} \rightarrow \text{Integer} \]

\[ \text{Ol.m: Boolean} \rightarrow \text{Boolean} \]
Example class OL I

Callgraph

\[ CG(\text{Main.main}: \text{Integer} \rightarrow \text{Integer}) \]
\[ = \]
\[ \text{Main.main}: \text{Integer} \rightarrow \text{Integer} \]
\[ \& \text{Boolean} \rightarrow \text{Boolean} \]
\[ \Downarrow \]
\[ \text{OL.m}: \text{Integer} \rightarrow \text{Integer} \]

\[ CG(\text{Main.main}: \text{Boolean} \rightarrow \text{Boolean}) \]
\[ = \]
\[ \text{Main.main}: \text{Integer} \rightarrow \text{Integer} \]
\[ \& \text{Boolean} \rightarrow \text{Boolean} \]
\[ \Downarrow \]
\[ \text{OL.m}: \text{Boolean} \rightarrow \text{Boolean} \]

Supertype

\[ \text{Integer} \rightarrow \text{Integer} \]
\[ \text{Boolean} \rightarrow \text{Boolean} \]
Example class OL II

Code generation

```java
class Main {
    Integer main(Integer x) {
        OL ol;
        ol = new OL();
        return ol.m(x); }

    Boolean main(Boolean x) {
        OL ol;
        ol = new OL();
        return ol.m(x);
    }
}
```

Code is unchanged in comparison to the first approach
**Example class Matrix**

**Callgraph** $\mathcal{CG}(\text{Matrix.mul}: \tau)$ for all $\tau$

```
Matrix.mul: Vector<Vector<Int>> -> Matrix &
Vector<Vector<Int>> -> Matrix &
& ... &
Matrix -> Vector<Vector<Int>>
```

```
Vector<T>.size: int
Vector<T>.elementAt: int -> T
Vector<T>.addElement: T -> void
```
Example class `Matrix` I

**Callgraph** $CG(\text{Matrix.mul}: \tau)$ for all $\tau$

```
Matrix.mul: Vector< Vector< Int>> -> Matrix &
Vector< Vector<Int>> -> Matrix &
& ... &
Matrix -> Vector< Vector< Int>>
```

**Supertype:**

```
Vector<? extends Vector<? extends Integer>> → Matrix
```
Example class Matrix II

Code generation (only one method!)

Matrix `mul(Vector<? extends Vector<? extends Integer>> m) {`
  Matrix ret = new Matrix();
  int i = 0;
  while(i < size()) {
    Vector<Integer> v1 = this.elementAt(i);
    Vector<Integer> v2 = new Vector<Integer>();
    int j = 0;
    while(j < v1.size()) {
      int erg = 0;
      int k = 0;
      while(k < v1.size()) {
        erg = erg + ...; k++;
      }
      v2.addElement(new Integer(erg)); j++;
    }
    ret.addElement(v2); i++;
  }
  return ret; }
}
The Algorithm

Input: A Java program $p$ with inferred (intersection) types.
Output: A Java program $p'$, where the methods have standard Java types. The semantics of $p$ and $p'$ are equal.

1. Step: For every class $cl$ in $p$ consider for each method $m$ the intersection type $ty_m$:
   - Build the callgraph $CG(\,cl.m : \tau\,)$ for each function type $\tau$ of the intersection type $ty_m$.
   - Group all elements $\tau$ of $ty_m$, where $CG(\,cl.m : \tau\,)$ is the same graph and there is a supertype.

2. Step: Determine the supertype of the respective group.

3. Step: Generate for each group of function types the corresponding Java code with the supertype as standard typing in $p'$.
Definition [Damas, Milner 1982]:

“A type-scheme for a declaration is a principal type-scheme, if any other type-scheme for the declaration is a generic instance of it.”
Definition [Damas, Milner 1982]:

“A type-scheme for a declaration is a principal type-scheme, if any other type-scheme for the declaration is a generic instance of it.”

Generalization to the Java type system [Plümicke 2007, PPPJ07]

“An intersection type-scheme for a declaration is a principal type-scheme, if any (non–intersection) type-scheme for the declaration is a subtype of a generic instance of one element of the intersection type-scheme.”
Example principal typing I

```java
import java.util.Vector;
import java.util.Stack;

class Put {
    <T> putElement(T ele, Vector<T> v) {
        v.addElement(ele);
    }

    <T> putElement(T ele, Stack<T> s) {
        s.push(ele);
    }

    main(ele, x) {
        putElement(ele, x);
    }
}
```
Example principal typing II

The inferred intersection type:

\[
\text{main} : T \times \text{Vector}\langle T \rangle \rightarrow \text{void} \& T \times \text{Stack}\langle T \rangle \rightarrow \text{void}.
\]

is a principal type.
Example principal typing II

The inferred intersection type:

$$\text{main} : T \times \text{Vector}<T> \rightarrow \text{void} \& T \times \text{Stack}<T> \rightarrow \text{void}.$$  

is a principal type.

But there is another principal type:

$$\text{main} : T \times \text{Vector}<T> \rightarrow \text{void},$$

as $\text{Stack}<T> \leq \ast \text{Vector}<T>$. 
class Put {
  <T> void putElement(T ele, Vector<T> v) { ... }

  <T> void putElement(T ele, Stack<T> s) { ... }

  <T> void main(T ele, Vector<T> x) {
    x.putElement(ele, x); }

  <T> void main(T ele, Stack<T> x) {
    x.putElement(ele, x); }
}
Example principal typing III

class Put {
    <T> void putElement(T ele, Vector<T> v) { ... }

    <T> void putElement(T ele, Stack<T> s) { ... }

    <T> void main(T ele, Vector<T> x) {
        x.putElement(ele, x); }

    <T> void main(T ele, Stack<T> x) {
        x.putElement(ele, x); }

The principal type

main : T × Vector<T> → void

is correct but not meaningful!!
Refined definition of *Principal typing*

“An intersection type-scheme for a declaration is a *principal type-scheme*, if any (non-intersection) type-scheme θ for the declaration is a subtype of a generic instance of one element of the intersection type-scheme τ and θ and τ have the same callgraph.”
“An intersection type-scheme for a declaration is a principal type-scheme, if any (non–intersection) type-scheme \( \theta \) for the declaration is a subtype of a generic instance of one element of the intersection type-scheme \( \tau \) and \( \theta \) and \( \tau \) have the same callgraph.”

This refined definition guarantees, that for each method, which is generated by the resolving algorithm, at least one typing is contained in the principal type.
Example Put (cont.)

\[ CG(\text{Put.main}: T \times \text{Vector}\langle T \rangle \rightarrow \text{void}) \neq CG(\text{Put.main}: \text{Integer} \times \text{Stack}\langle \text{Integer} \rangle \rightarrow \text{void}) \]

\[ = \]

\[ \text{Put.main}: T \times \text{Vector}\langle T \rangle \rightarrow \text{void} \]
\[ \& T \times \text{Stack}\langle T \rangle \rightarrow \text{void} \]

\[ \downarrow \]

\[ \text{Put.putElement}: T \times \text{Vector}\langle T \rangle \rightarrow \text{void} \]

\[ \text{main}: T \times \text{Vector}\langle T \rangle \rightarrow \text{void} \text{ is no principal type, but} \]
\[ \text{main}: T \times \text{Vector}\langle T \rangle \rightarrow \text{void} \& T \times \text{Stack}\langle T \rangle \rightarrow \text{void} \text{ is a principal type.} \]
Conclusion and Outlook

Conclusion

- Semantics for Java methods with intersection types
- Resolving algorithm of intersection types
- Code generation for methods with intersection types possible
  ⇒ type selection is not longer necessary in the Eclipse plugin.
- (Redefined) Principal type property
Conclusion and Outlook

Conclusion

- Semantics for Java methods with intersection types
- Resolving algorithm of intersection types
- Code generation for methods with intersection types possible
  - type selection is not longer necessary in the Eclipse plugin.
- (Redefined) Principal type property

Outlook

At the moment: Type inference algorithm infers typings, which are later erased as subtypes by the resolving algorithm.

Purpose: Type inference algorithm infers only supertypes, such that no typings are erased in the resolving algorithm.