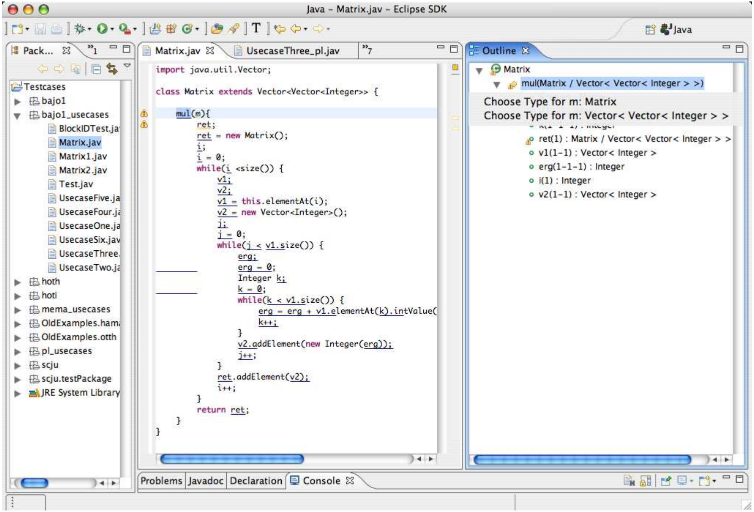
Resolving of Intersection Types in Java

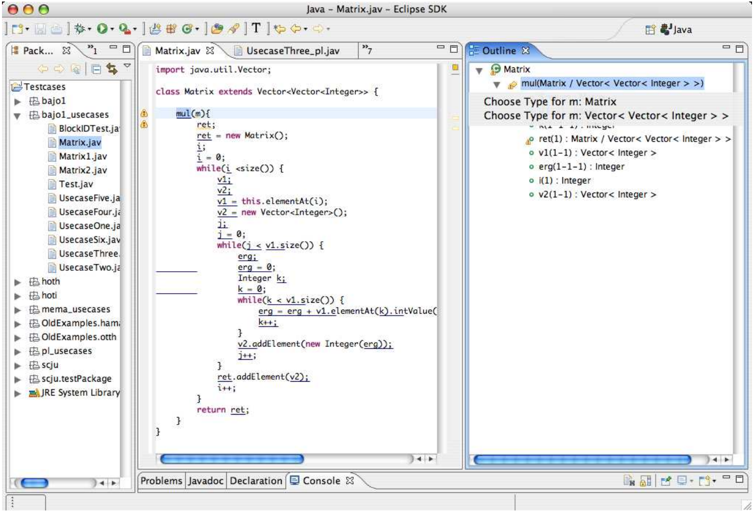
Martin Plümicke

University of Cooperative Education
Stuttgart/Horb

5. Mai 2008

Overview


Java – Matrix.jav – Eclipse SDK
Java
Pack...
Testcases
bajo1
bajo1_usecases
BlockIDTest.ja
Matrix.jav
Matrix1.jav
Matrix2.jav
Test.jav
UsecaseFive.ja
UsecaseFour.ja
UsecaseOne.ja
UsecaseSix.jav
UsecaseThree.
UsecaseTwo.ja
hoth
hoti
mema_usecases
OldExamples.ham
OldExamples.otth
pl_usecases
scju
scju.testPackage
JRE System Library
Matrix.jav
UsecaseThree_pl.jav
import java.util.Vector;
class Matrix extends Vector<Vector<Integer>> {
mul(m){
ret;
ret = new Matrix();
i;
i = 0;
while(i <size()) {
v1;
v2;
v1 = this.elementAt(i);
v2 = new Vector<Integer>();
j;
j = 0;
while(j < v1.size()) {
erg;
erg = 0;
Integer k;
k = 0;
while(k < v1.size()) {
erg = erg + v1.elementAt(k).intValue(
k++;
}
v2.addElement(new Integer(erg));
j++;
}
ret.addElement(v2);
i++;
}
return ret;
}
}
Outline
Matrix
mul(Matrix / Vector< Vector< Integer > >)
Choose Type for m: Matrix
Choose Type for m: Vector< Vector< Integer > >
ret(1) : Matrix / Vector< Vector< Integer > >
v1(1-1) : Vector< Integer >
erg(1-1-1) : Integer
i(1) : Integer
v2(1-1) : Vector< Integer >
Problems
Javadoc
Declaration
Console



Purpuse: Byte-code generation for methods with intersection types

Semantics of type–inferred Java programs

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

Semantics of type–inferred Java programs

- 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

```
... 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
- `t1 , ... , tn` have unambigous types during execution
- by the argument types 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);
```

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); 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

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

Result for `Main`:

```
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

$$\texttt{mul:}\ \&_{\beta,\alpha}(\beta \rightarrow \alpha),$$

where

$\beta \leq^* \texttt{Vector<? extends Vector<? extends Integer>>}$,
$\texttt{Matrix} \leq^* \alpha$

Example: Multiplication of matrices II

$$\texttt{mul}: \&_{\beta,\alpha}(\beta \rightarrow \alpha),$$

where

$\beta \leq^{*}$ `Vector<? extends Vector<? extends Integer>>`,
`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

$$\texttt{mul:}\ \&_{\beta,\alpha}(\beta \rightarrow \alpha),$$

where

$\beta \leq^* \texttt{Vector<? extends Vector<? extends Integer>>}$,
$\texttt{Matrix} \leq^* \alpha$

```
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) { ... }
```

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

$$\mathcal{CG}(\, cl.m : \tau \,)$$

Callgraph of the method m in the class cl with the typing τ.

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

$$\mathcal{CG}(\ cl.m : \tau\ )$$

Callgraph of the method m in the class cl with the typing τ.

Supertype of function types: Subtyping ordering
$\theta_i \leq^* \theta'_i, \theta \leq^* \theta' \Rightarrow$

$$\theta_1 \times \ldots \times \theta_n \to \theta' \leq^* \theta'_1 \times \ldots \times \theta'_n \to \theta$$

Example class OL I

Callgraph

$\mathcal{CG}(\texttt{Main.main} : \texttt{Integer} \rightarrow \texttt{Integer})$
$=$

```
Main.main: Integer->Integer
    & Boolean -> Boolean
```

$\downarrow$

```
OL.m: Integer -> Integer
```

$\mathcal{CG}(\texttt{Main.main} : \texttt{Boolean} \rightarrow \texttt{Boolean})$
$=$

```
Main.main: Integer->Integer
    & Boolean -> Boolean
```

$\downarrow$

```
Ol.m: Boolean -> Boolean
```

Example class OL I

Callgraph

$\mathcal{CG}(\,\mathtt{Main.main} : \mathtt{Integer} \to \mathtt{Integer}\,)$

$=$

Main.main: Integer->Integer & Boolean -> Boolean

$\downarrow$

OL.m: Integer -> Integer

$\mathcal{CG}(\,\mathtt{Main.main} : \mathtt{Boolean} \to \mathtt{Boolean}\,)$

$=$

Main.main: Integer->Integer & Boolean -> Boolean

$\downarrow$

Ol.m: Boolean -> Boolean

Supertype

$\mathtt{Integer} \to \mathtt{Integer}$

$\mathtt{Boolean} \to \mathtt{Boolean}$

Example class OL II

Code generation

```
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` I

Callgraph $\mathcal{CG}($ `Matrix.mul` $: \tau\,)$ for all τ

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

Vector<T>.size:-> int   Vector<T>.elementAt -> T   Vector<T>.addElement: T -> void
```

Example class `Matrix` I

Callgraph $\mathcal{CG}($ `Matrix.mul` $: \tau\,)$ for all τ

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

```
Vector<T>.size:-> int   Vector<T>.elementAt -> T   Vector<T>.addElement: T -> void
```

Supertype:

$$\texttt{Vector<? extends Vector<? extends Integer>>} \longrightarrow \texttt{Matrix}$$

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 $\mathcal{CG}(\ cl.m : \tau\ )$ for each function type τ of the intersection type ty_m.
    - Group all elements τ of ty_m, where $\mathcal{CG}(\ cl.m : \tau\ )$ 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'.

Principal Typing

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."

Principal Typing

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 (old definition)

"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

```
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) {
      x.putElement(ele, x);
    }
}
```

Example principal typing II

The inferred intersection type:

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

This is a principal type.
But there is another principal type:

$$\mathtt{main} : \mathtt{T} \times \mathtt{Vector<T>} \rightarrow \mathtt{void}.$$

Example principal typing II

The inferred intersection type:

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

This is a principal type.
But there is another principal type:

$$\texttt{main} : \texttt{T} \times \texttt{Vector<T>} \rightarrow \texttt{void}.$$

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."

Example Put (cont.)

$$\mathcal{CG}(\,\texttt{Put.main} : \texttt{T} \times \texttt{Vector<T>} \longrightarrow \texttt{void}\,) \neq \mathcal{CG}(\,\texttt{Put.main} : \texttt{Integer} \times \texttt{Stack<Integer>} \longrightarrow \texttt{void}\,)$$

$$=$$

```
Put.main: T x Vector<T> -> void
     & T x Stack<T> -> void
```

$$\downarrow$$

```
Put.putElement: T x Vector<T> -> void
```

$$=$$

```
Put.main: T x Vector<T> -> void
     & T x Stack<T> -> void
```

$$\downarrow$$

```
Put.putElement: T x Stack<T> -> void
```

$\texttt{main} : \texttt{T} \times \texttt{Vector<T>} \longrightarrow \texttt{void}$ is no principal type, but

$\texttt{main} : \texttt{T} \times \texttt{Vector<T>} \longrightarrow \texttt{void}\ \&\ \texttt{T} \times \texttt{Stack<T>} \longrightarrow \texttt{void}$ 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
- (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
- (Redefined) Principal type property

Outlook

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

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