

Resolving of Intersection Types in Java

Martin Plümicke

University of Cooperative Education Stuttgart
Department of Information Technology
Florianstraße 15, D-72160 Horb
m.pluemicke@ba-horb.de

Abstract. In the past we analyzed typeless Java programs. One of our results was, that there may be different correct typings for one method. This means that the principal types of such methods are intersection types. We presented a type-inference algorithm. For typeless Java methods the algorithm infers its principal intersection type. Unfortunately, like Java byte-code, Java does not allow intersection types.

In this paper we present an algorithm, which resolves intersection types of Java methods, such that Java programs with standard typings are generated.

Additionally, we will refine the definition of Java method principal types.

1 Introduction

In [7, 6] we presented a type inference algorithm for a core Java language. The algorithm allows us to write typeless Java programs. The algorithm determines all correct possible typings. One of the results of [7] is that typeless Java methods can have more than one correct typing, which means that the method is typed by an intersection of function types. The type inference algorithm infers this intersection type.

We have implemented the algorithm as an Eclipse plugin. As the Java byte-code does not allow intersection types, we implemented the plugin, such that after type inference the user has to select one of the possible typings. The byte-code is generated only for this selected typing.

It is our purpose to improve the plugin such that type selection is no longer necessary. This means that byte-code can be generated for methods with inferred intersection types.

For this paper we need to consider some important definitions from [6]. Let $\text{SType}_{TS}(BTV)$ be the set of usual Java types. In the formal definition ([6], Def. 3) BTV stands for the set of (bounded) type variables ([6], Def. 1). We call the elements of $\text{SType}_{TS}(BTV)$ *simple types*. In Java programs simple types describe types of fields, types of methods' parameters, return types of methods, and types of local variables. For the type inference we need intersections of function types, which describe the types of methods. A *function type* is given as $\theta_1 \times \dots \times \theta_n \rightarrow \theta_0$, where $\theta_i \in \text{SType}_{TS}(BTV)$. An *intersection type* is an intersection $ty_1 \& \dots \& ty_m$ of function types ty_1, \dots, ty_n . There is a *subtyping*

ordering on simple types. We denote this by \leq^* (cp. [6], Def. 5). Finally there are arguments of type terms, which are *wildcard types*. There are two kinds of wildcard types “? extends *ty*” respectively “? super *ty*”. We abbreviate them by $?ty$ respectively by $?ty$. For further definitions we refer to [6], Section 2.

The type inference algorithm determines types of methods, which are given as **Java** code without types. The inferred method types probably contain intersection types. Unfortunately, standard **Java** does not include intersection types for methods. This means that there is no canonical strategy to generate byte-code for methods with intersection types. As there is an isomorphic mapping from the signature of the standard **Java** methods to the signature of the corresponding methods in the byte-code, the intersection types must be resolved for code generation. In this paper we present the resolving of intersection types by transforming the **Java** code with inferred intersection types to equivalent standard **Java** code (without intersection types).

The paper is organized as follows. In the next section we define the semantics of method calls for methods with intersection types. In the third section we give a first approach to resolve intersection types. We will see that this approach leads to incorrect and inefficient **Java** programs. In the fourth section we consider call-graphs of **Java** methods. Call-graphs are the base of our intersection type resolving algorithm. We present the corresponding algorithm in the fifth section. In the sixth section we give a refined definition for **Java** method’s principal types, which is based on the ideas of intersection type resolving. Then we consider related work on intersection types and principal typings. Finally we close with a summary and an outlook.

2 Semantics of type-inferred Java programs

The semantics of a type-inferred **Java** program is defined straightforwardly. All control-structures have the same semantics as in standard **Java** (e.g. [4], [1]). Only the semantics of the method calls differ, as there are intersection types.

The main idea to define the semantics of method calls for methods with intersection types is the following: One typing of the intersection type of the method is determined by the argument types of the method calls. The method is then executed with this typing.

Definition 1 (Semantics of method calls). *Let a Java method m be given, where one typing of its intersection type is selected. This means that the method corresponds to a standard typed Java method.*

$$\dots m (\dots) \{ \dots receiver.method(t1, \dots, tn); \dots \}$$

The method m contains a method call $receiver.method(t1, \dots, tn)$, where $receiver$ has the type $recty$ and $t1 \dots tn$ have the types $ty_1 \dots ty_n$, respectively. Furthermore the result of the method call has the type $retty$.

Then, the smallest class is determined, which is a supertype from $recty$ with the method $method$, where $ty'_1 \times \dots \times ty'_n \rightarrow ty'$ is one element of its intersection

type and for $1 \leq i \leq n$ the types ty'_i are the smallest supertypes of ty_i , respectively, and ty' is a subtype of $rettype$. Then this method *method* is executed with the corresponding typing.

Example 1. Let the following typeless Java program be given:

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

By type-inference the following typings are determined:

```
OL.m : Integer → Integer
OL.m : Boolean → Boolean
Main.main : Integer → Integer & Boolean → Boolean
```

Let the Java program be extended by the simple class `simpleClass` with the method `new_meth`:

```
class simpleClass {
    new_meth(x) {
        Main rec = new Main();
        Integer r = rec.main(x); } }
```

We consider the method call `rec.main(x)`. The typing is given as: `rec:Main`, `x:Integer`, and `main:Integer → Integer`.

The class `Main` itself is the smallest class, which is a supertype of `Main` with the method `main`, where `Integer → Integer` is one element of its intersection type, `Integer` is the smallest supertype of `Integer`, and the return type `Integer` of `main` is also a subtype of the demanded type `Integer`.

This means that the `main` method in the class `Main` is called with the following typing:

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

3 First approach

As a first approach to resolve inferred intersection types a new overloaded Java method with standard typing is generated for each element of the intersection type.

Considering the following examples, we will recognize, that this strategy works only in some cases.

Example 2. Let the Java program from Example 1 be given again. As said above the following typings are determined by type-inference:

```

OL.m : Integer → Integer
OL.m : Boolean → Boolean
Main.main : Integer → Integer & Boolean → Boolean

```

If we generate an own Java method for each element of the intersection type we get the main class:

```

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); } }

```

The result is a correct Java program.

The next example shows that this strategy does not work in all cases.

Example 3. Let the following Java program be given, which implements the multiplication of two integer matrices.

```

class Matrix extends Vector<Vector<Integer>> {
    mul(m){
        ret; ret = new Matrix();
        Integer i = 0;
        while(i <size()) {
            v1; v1 = this.elementAt(i);
            v2; v2 = new Vector<Integer>();
            Integer j = 0;
            while(j < v1.size()) {
                Integer erg = 0;
                Integer 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; }
    }

```

By type-inference the following typings are determined: $\text{mul} : \&_{\beta, \alpha}(\beta \rightarrow \alpha)$, where

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

This means, that for each pair of arguments and result types a new method `mul` would be generated:

```

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) { ... } }

```

This is not a correct Java program. There are two conflicts. On the one hand it is not allowed for overloaded methods that argument types have different instances of the same generic type. For example this means for the method `mul`, that the argument type `Vector<?Vector<Integer>>` of one declaration is not allowed, if there is another method declaration with the argument type `Vector<?Vector<?Integer>>`. This is no type theoretical problem. But as the byte-code works only with raw types, there is no difference between both argument types during execution.

On the other hand different method declarations of `mul`, where the argument type is in all declarations equal, e.g. `Vector<Vector<Integer>>`, lead to ambiguity.

If we consider the generated method declarations more accurately, we recognize that in all declarations the same code is executed. One approach to solve the problem could be to generate one declaration with a supertype of all other declarations: `mul : Vector<?Vector<?Integer>> → Matrix`. The argument type `Vector<?Vector<?Integer>>` is the supertype of all other argument types and `Matrix` is the subtype of all other result types. Later on, we will call this function type the supertype of all possible typings.

Now, we will present the algorithm, which resolves the intersection types, such that incorrect and unnecessary copies of methods are avoided.

4 Call-graph

We consider call-graphs of Java methods as the base of the intersection type resolving algorithm. Call-graphs are graphs of method declarations, which contain all methods, that are called during the execution, for a given method with one typing.

Definition 2 (Call-graph). *Let p be a Java program containing one or more classes, where the (intersection) types of the methods are inferred. Furthermore let the triple $cl.m : \tau$ be given, where m is a declared method in the class cl and τ is an instance of one element of the inferred intersection type. The call-graph $\mathcal{CG}(cl.m : \tau)$ is given as the pair (M, MC) , where M is the set of declared methods. $MC \subseteq M \times M$ is given as the smallest set with the following properties:*

- *Let $cl.m : ty \in M$, where the function type τ is an instance of an element of the intersection type ty . It holds $(cl.m : ty, cl'.m' : ty') \in MC$ for all methods m' , which are called in $cl.m : ty$, if m has the function type τ .*

- If $(cl.m : ty, cl'.m' : ty') \in MC$ and $cl'.m' : ty'$ is called with function type τ' , then $(cl'.m' : ty', cl''.m'' : ty'') \in MC$ for all methods m'' , which are called in $cl'.m' : ty'$, if m' has the function type τ' .

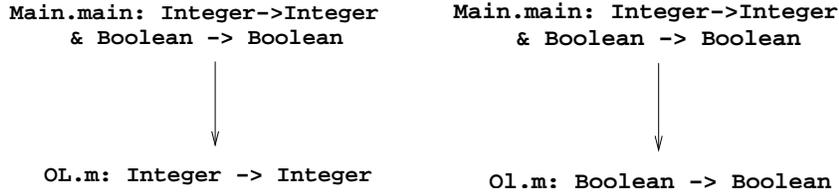


Fig. 1. $CG(\text{Main.main} : \text{Integer} \rightarrow \text{Integer})$, $CG(\text{Main.main} : \text{Boolean} \rightarrow \text{Boolean})$

Example 4. Let the Java program from Example 1 be given again. From the inferred typings the following call-graphs are determined:

$CG(\text{Main.main} : \text{Integer} \rightarrow \text{Integer})$ is the left call-graph in Fig. 1. In this case $\tau = \text{Integer} \rightarrow \text{Integer}$. In Main.main with the type τ the method m with function type $\text{Integer} \rightarrow \text{Integer}$ is called.

$CG(\text{Main.main} : \text{Boolean} \rightarrow \text{Boolean})$ is the right call-graph in Fig. 1. In this case $\tau = \text{Boolean} \rightarrow \text{Boolean}$. In Main.main with the type τ the method m with function type $\text{Boolean} \rightarrow \text{Boolean}$ is called. It is obvious that for each different type of main different methods are called.

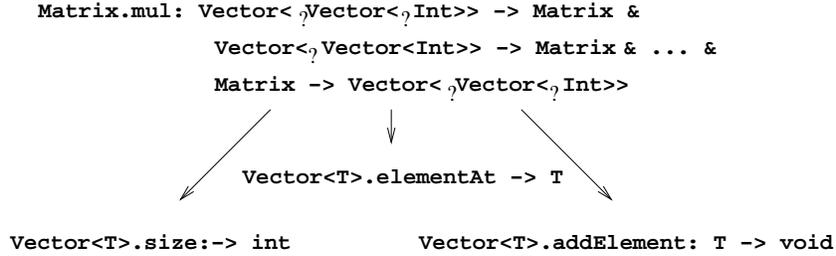


Fig. 2. The call-graph $CG(\text{Matrix.mul} : \tau)$ for all τ

Example 5. Let us consider again the Java program `Matrix` (Example 3). In this case for all function types τ of the triple $\text{Matrix.mul}:\tau$ the call-graphs $CG(\text{Matrix.mul} : \tau)$ are the same. The call-graph is given in Fig. 2.

In the two given examples the inferred types contain no generics. Therefore no instances of elements of intersection types are considered. The next example presents a call-graph of an instance of an element of an intersection type.

Example 6. Let the following Java program be given:

```
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); } }
```

The inferred intersection type of `main` is

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

The call-graph $\mathcal{CG}(\text{Put.main} : \text{Integer} \times \text{Stack}\langle \text{Integer} \rangle)$ is given as:

$$\begin{array}{c} \text{Put.main} : T \times \text{Vector}\langle T \rangle \rightarrow \text{void} \\ \quad \& \ T \times \text{Stack}\langle T \rangle \rightarrow \text{void} \\ \downarrow \\ \text{Put.putElement} : T \times \text{Stack}\langle T \rangle \rightarrow \text{void} \end{array}$$

As $\text{Integer} \times \text{Stack}\langle \text{Integer} \rangle \rightarrow \text{void}$ is an instance of $T \times \text{Stack}\langle T \rangle \rightarrow \text{void}$ in `main` the method `putElement: T × Stack<T> → void` is called.

The following *intersection type resolving algorithm* bases on the call-graph.

5 The resolving algorithm

In this section we describe the algorithm, which resolves the intersection types, such that standard Java programs with standard types are generated.

Before we can present the algorithm we have to generalize the definition of the subtyping ordering to function types.

Definition 3 (Subtyping on function types). *Let the subtyping ordering \leq^* on simple types ([6], Def. 5) be given. For two function types $ty_1 = \theta_1 \times \dots \times \theta_n \rightarrow \theta'$ and $ty_2 = \theta'_1 \times \dots \times \theta'_n \rightarrow \theta$ holds: ty_1 is a subtype of ty_2 if for $1 \leq i \leq n$ holds $\theta_i \leq^* \theta'_i$, respectively, and $\theta \leq^* \theta'$. We call the maximum in the subtyping ordering supertype.*

Example 7. The function type `Matrix → Vector<? extends Vector<Integer>>` is a subtype of `Vector<Vector<Integer>> → Matrix` as it holds

$$\text{Matrix} \leq^* \text{Vector}\langle \text{Vector}\langle \text{Integer} \rangle \rangle \text{ and} \\ \text{Matrix} \leq^* \text{Vector}\langle ? \text{ extends Vector}\langle \text{Integer} \rangle \rangle.$$

In contrast `Integer → Integer` is no subtype of `Number → Number`, as `Integer \leq^* Number`, but not vice versa.

Now we are able to present the algorithm.

The algorithm

Input: A Java program p consisting of different classes with inferred (intersection) types of its methods.

Output: A Java program p' consisting of the same classes as 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 call-graph $\mathcal{CG}(cl.m : \tau)$ 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' .

Example 8. Let us consider the Java program from Example 1 as input p .

1. **Step:** The only method, where an intersection type is inferred, is `main` in the class `Main`. The intersection type is given as

$$\text{Integer} \rightarrow \text{Integer} \ \& \ \text{Boolean} \rightarrow \text{Boolean}.$$

- The call-graphs are given in Example 4.
- There are two groups, each with one element:
 - { `Main.main : Integer → Integer` } and
 - { `Main.main : Boolean → Boolean` }, as there are different call-graphs.

2. **Step:** The respective superotypes are also

$$\text{Integer} \rightarrow \text{Integer} \ \text{and} \ \text{Boolean} \rightarrow \text{Boolean}.$$
3. **Step:** The corresponding Java code, which is added to p' is given as

```

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); }

```

If we consider the result it is obvious that the result is the same as in the first approach (cp. Example 2). In the following example we will see some differences in comparison to the first approach.

Example 9. Let the Java program `Matrix` from Example 3 be given as input.

1. **Step:** The intersection type of `mul` is given as: $\text{mul} : \&_{\beta, \alpha}(\beta \rightarrow \alpha)$, where

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

$$\text{Matrix} \leq^* \alpha$$

- The call-graph for all elements τ of the inferred intersection type of `mul` is given in Fig. 2.

- As there is a supertype $\text{Vector}\langle ? \text{Vector}\langle ? \text{Integer} \rangle \rangle \rightarrow \text{Matrix}$ for all elements, there is only one group, which contains all elements τ .
- 2. Step:** The supertype is determined as: $\text{Vector}\langle ? \text{Vector}\langle ? \text{Integer} \rangle \rangle \rightarrow \text{Matrix}$.
- 3. Step:** The corresponding Java code, which is added to p' is given as:

```
Matrix mul(Vector<? extends Vector<? extends Integer>> m) { ... }
```

The example shows that the algorithm solves the problems, which arose in the first approach (cp. Example 3).

The following theorem shows the correctness of the algorithm.

Theorem 1. *Let p be a Java program and p' be the result of applying the intersection type resolving algorithm. The semantics of p equals to the semantics of p'*

6 Principal type

In the following we will refine the definition of principal types (Def. 8 [6]) for Java methods.

Definition 4 (Principal type). *An intersection type of a method*

$$m : (\theta_{1,1} \times \dots \times \theta_{1,n} \rightarrow \theta_1) \& \dots \& (\theta_{m,1} \times \dots \times \theta_{m,n} \rightarrow \theta_m)$$

in a class $C1$ is called principal if for any correct type annotated method declaration $\text{rty } m(\text{ty1 } a_1, \dots, \text{ty}_n \text{ an}) \{ \dots \}$ there is an element $(\theta_{i,1} \times \dots \times \theta_{i,n} \rightarrow \theta_i)$ of the intersection type and there is a substitution σ , such that $\sigma(\theta_i) \leq^ \text{rty}$, $\text{ty1} \leq^* \sigma(\theta_{i,1})$, \dots , $\text{ty}_n \leq^* \sigma(\theta_{i,n})$ and*

$$CG(C1.m : \theta_{i,1} \times \dots \times \theta_{i,n} \rightarrow \theta_i) = CG(C1.m : \text{ty1} \times \dots \times \text{ty}_n \rightarrow \text{rty})$$

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 10. In Example 6 the principal type of `main` is

```
main : T × Vector<T> → void & T × Stack<T> → void.
```

7 Related Work

Besides our introduction of intersection types for methods in Java with type inference, in standard Java there are intersection types of simple types during compile-time [4], §4.9). Intersection types arise in the processes of capture conversion and type inference during method invocation. In Java it is not allowed to write an intersection type as a part of a program.

Basically, the intersection type discipline was introduced by Coppo and Dezani [2]. In the type system of Damas and Milner [3] some λ -terms are not typable. Therefore the type system is extended by intersection types. The type inference problem for these type systems is not decidable in general (e.g. [5]). Our Java

type system is a restriction of them. In comparison, our Java type system contains no λ -terms. This means that we do not have the function type constructor \rightarrow and no higher-order functions. Instead of that, Java has the function template $(ty_1, \dots, ty_n) \rightarrow ty$.

In [8] a general definition of principal type property is given. Our definition (Def. 4) as well as the definition of Damas and Milner [3] satisfies the definition of [8].

8 Conclusion and Outlook

Beginning with the result of [7, 6] that the inferred principal types of typeless Java methods can be intersection types, we showed how they can be resolved. This means that type inferred Java programs with intersection types can be translated in Java byte-code. In the Eclipse plugin type selection is no longer necessary.

Some properties of the resolving algorithm lead to a refined definition of a Java method's principal type. The refined version guarantees that for each method, which is generated by the resolving algorithm, at least one element is contained in the principal type.

Further investigation is necessary to optimize the procedure: *type inference algorithm*, *intersection type resolving algorithm*, and *code generation*. At the moment the type inference algorithm infers some types for the methods, which are erased again in the second step of the resolving algorithm.

References

1. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer, 1999.
2. M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ -calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.
3. L. Damas and R. Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.
4. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The JavaTM Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.
5. D. Leivant. Polymorphic type inference. *Proc. 10th Symposium on Principles of Programming Languages 1982*, 1983.
6. M. Plümicke. Typeless Programming in Java 5.0 with wildcards. In V. Amaral, L. Veiga, L. Marcelino, and H. C. Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, ACM International Conference Proceeding Series, pages 73–82, September 2007.
7. M. Plümicke and J. Bäuerle. Typeless Programming in Java 5.0. In R. Gitzel, M. Aleksey, M. Schader, and C. Krintz, editors, *4th International Conference on Principles and Practices of Programming in Java*, ACM International Conference Proceeding Series, pages 175–181. Mannheim University Press, August 2006.
8. S. van Bakel. Principal type schemes for the strict type assignment system. *Journal of Logic and Computing*, 3(6):643–670, 1993.