Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

Java type inference as an Eclipse plugin

Andreas Stadelmeier[1]

Abstract: While in Java 5.0 generics have been introduced in Java 8 the language has been expanded by lambda expressions. The very popular feature in functional programming languages, type inference, is introduced in Java, but only in a very restricted form, similar as in other object-oriented languages. This paper presents an Eclipse plugin which allows writing Java 8 programs without any type annotation. A type inference algorithm determines all possible types which the Eclipse plugin presents to the user afterwards. So the user can select the desired type. This plugin allows to determine most general types, which are often not obvious. The programs therefore become more reuseable, which is the goal of any software development.

Keywords: Code generation, language design, program design and implementation, type inference, type system

1 Introduction

Java has been extended by two features that are well-known from functional programming languages. In Java 5.0 generics have been introduced while in Java 8 lambda expressions have been established. A further very convenient feature from functional programming languages, the type inference, is included only in a very restricted form. Let us consider the following example.

```
interface Fun1<R, T> { R apply(T arg); }
interface Fun2<R, T1, T2> {R apply(T1 arg1, T2 arg2); }

class Matrix {
  Fun1<Fun1<Matrix,Fun2<Matrix,Matrix,Matrix>>,Matrix>
    op = (m) -> (f) -> f.apply(this, m);
}
```

The function `op` applies a given function `f` (second argument) by the method `apply` to `this` and `m` (first argument).

[1] Baden-Württemberg Cooperative State University Stuttgart, Department of Computer Science, Florianstraße 15, D–72160 Horb, a.stadelmeier@hb.dhbw-stuttgart.de

If we consider the declaration of `op` we recognize that the arguments `m` and `f` are untyped. This is possible due to Java 8's restricted type inference system. But the variable `op` must be typed. As a lambda expression is assigned to `op` and as no real function-types are included in Java 8, a functional interface is the compatible target type of `op`. In our example we declare parametrized functional interfaces `Fun1` and `Fun2`, which correspond to real function-types.

If we consider the function `op` in detail we will see that the given type is not the most general. The type

```
<A,B>[2]Fun1<Fun1<B, Fun2<B, Matrix, A>>, A>
```

is also correct but it is more general. Unfortunately, there are further correct typings, where it is not obvious, which is the most general one.

Our type inference system allows to declare

```
class Matrix extends Vector<Vector<Integer>> {
    op = (m) -> (f) -> f.apply(this, m);
}
```

where no additional functional interfaces must be declared and no type for `op` must be given. The system determines all most general types[3] as functional interfaces of the form `Fun`*N*. As there are more than one most general type our Eclipse plugin allows the user to select the favored type.

This approach supports the programmer significantly, as no functional interfaces as target types of lambda expressions have to be defined, types of complex expressions with generics and wildcards need not to be detected and writing reusable code is assisted by providing most general types.

At the moment, object-oriented languages mostly use local type inference (e.g. Scala [Od14]). Local type inference determines types by a combination of local type propagation, from adjacent nodes in the syntax tree, and local constraint solving, rather than by global constraint solving [PT00, OZZ01]. In languages with local type inference only a few type annotations can be omitted.

Our approach is based on global constraint solving and allows to omit all type annotations. Nevertheless the type inference system is a practical application, as the user is supported by an intelligent Eclipse plugin.

[2] The declaration of generic fields is not allowed in Java. In our extension we allow this, if the field represents a function given by a lambda expression.

[3] A type is considered most general when there is no type with a more general meaning. But often there are multiple most general types (comparable to multiple minimal elements in a partial ordering, which are not a minimum). A most general type is therefore normally not a principal type.

The paper is organized as follows. The next section gives a brief overview of type inference following the description of our type inference algorithm. The third section presents our Eclipse plugin followed by a summary and an outlook at the very end.

2 Type inference

Since the eighties much research has been done on type inference in programming languages. The base of nearly all type inference algorithms is the algorithm $\mathscr{W}$, that was given by Damas and Milner for ML [DM82].

The Hindley–Milner type inference algorithm determines for an untyped program a principal type. Principal type means that there are other correct typings, which are less general. In the Hindley–Milner approach less general means, there is substitution, that maps the principal type to the less general type.

For object-oriented programming languages with subtyping and parametric polymorphic types (generics) *local type inference* [PT00, OZZ01] is used. Local type inference is given by two techniques: First, type parameters in a function application are inferred from the function's value parameters by solving a constraint system. Second, it propagates known types down the syntax tree in order to infer some types of formal value parameters and provide additional guidance to type parameter inference. The Java type inference is a form of local type inference, where, however, some actually unnecessary type annotations are demanded. Consequently Scala uses local type inference.

2.1 Related Work

We considered until now type inference for different Java versions. In [Pl07] we presented a type inference algorithm for Java 5.0 with generics and wildcards. In [Pl11] we gave a type inference algorithm for Java with lambda expressions and real function types, that was a first idea for Java 8 [La10]. The type inference algorithm, which the Eclipse plugin is based upon, presented in this paper, is an improved extension of [Pl11] adapted to the functional interfaces of Java 8 [Pl14].

2.2 Comparision with other type inference algorithms

Type inference is already available in Java and other similiar programming languages. This chapter shows the differences between our plugin and other type inference systems which seem analogical to our approach.

2.2.1 Comparision with local type inference in Java 8

Java 8 is able to infer the parameter types of a lambda expression. Also the type of a lambda expression itself is inferred by the compiler. This is possible due to the fact that

there is a target type of the lambda expression in the environment. If a lambda expression is assigned to a variable or passed as a parameter the compiler uses the target type to determine the type of the lambda expression. Subsequently, the parameter types as well as the return type of the lambda expression are provided by the target type.

This does not work in case a method is overloaded with two functional interfaces, which are structurally the same. In Fig. 1 the interfaces `F1` and `F2` are structurally the same. The Java compiler is not able to determine which method has to be invoked: `m(F1 p)` or `m(F2 p)`.

```
interface F1{String apply();}
interface F2{String apply();}

class Test{
    String m(F1 p){ return p.apply(); }
    void m(F2 p){ }
    String main(){ return m(()->{return "Test";}); }
}
```

Fig. 1: Compilation error with Java 8

The global type inference algorithm described in this paper is able to infer the right type for the lambda expression in Fig. 1. It considers more than the target type to determine the missing type and is able to include and use more of the given information. Due to the fact that the `main` method returns a `String` and just one of the overloaded `m` methods does as well, only the type `F1` suits for the lambda expression in this context.

2.2.2 Comparision with local type inference in Scala

In Scala the local type inference [PT00, OZZ01] is implemented. Local type inference often allows omitting types of variable declarations and returns types of methods. Furthermore the types of instantiated generic classes or polymophic methods can be inferred, too. In the Scala tutorial[4] the mechanism is explained. See some examples in the following:

```
val x = 1 + 2 * 3        //the type of x is Int
val y = x.toString()     //the type of y is String
def succ(x: Int) = x + 1 //method succ returns Int values

class MyPair[A, B](x: A, y: B);

val p = MyPair(1, "scala") // type: MyPair[Int, String]
```

In contrast recursive defined functions must be typed explicitly. The following example would cause an error:

```
def fac(n: Int) = if (n == 0) 1 else n * fac(n - 1)
```

[4] `http://docs.scala-lang.org/tutorials/tour/local-typeinference.html`

In contrast to the considered type inference approaches for Java 8 and Scala in the approach presented by this paper all types can be inferred, automatically.

2.2.3 Comparision with Xtend

The Eclipse project also worked on a way to improve the Java programming language and created their own programming language named Xtend [Xt15]. Xtend is a modified version of Java. One of its main features is the possibility to omit type declarations in the code. The Xtend compiler inserts missing types into the source code before the actual compilition through the Java compiler. Therefore Xtend uses a local type inference algorithm which behaves similiar to the type inference used in Java 8. This approach of using a type inference algorithm to produce a less noisy Java is similiar-sounding to the type inference plugin described in this paper. But a comparision between Xtend and our Eclipse plugin shows the differences and capabilities of the respective type inference algorithms.

Xtend is not capable of determining types of recursive method calls. In the following example the type of *method* can not be determined by Xtend wherefore *Object* is assigned as the return type.

```
def method()
    if(...) return 1;
    else return test2();
```

Additionally Xtend is bound to a few constraints when it comes to infer the type of a local variable. Local variables must be declared with a initial value and only the type of the initial value is considered by the type inference algorithm of Xtend. Therefore the following Xtend code snippet generates a type error.

```
var variable = new Integer(1);
    variable = new Object(); //Error: cannot convert from Object to Integer
```

Our Eclipse plugin is able to infer the usecases given above due to a global type inference algorithm. Another difference between those two Java derivatives is the amount of modifications and extensions of the Java language. Xtend brings new keywords and the ability to omit unnecessary keywords and types. Our subset of Java sticks to the Java syntax and only adds the functionality of a global type inference algorithm. It supports the programmer in finding the most general type and writing less redundant and more reusable code.

3 Type inference algorithm

This chapter descripes our type inference algorithm which is also used by the Eclipes plugin. First a outline of the procedure of the algorithm is given following the introduction of relevant data structures.

3.1 Procedure of the global type inference algorithm

The procedure of the type inference algorithm consists out of the following parts.

Gathering constrains: A deep search algorithm crawls through the abstract syntax tree. It completes the tree by adding a type variable (type placeholder) for each missing type. Additional it generates constraints for every instruction and expression.

Unify: The algorithm described in [Pl09] unifies the gathered constraints. This process either succeeds by returning a set of unified type constraints or fails returning nothing.

Instantiating a solution: The type unification has multiple solutions. Each solution represents a possible replacement of the type placeholders with actual types. Therefore one solution has to be selected and instantiated to the type placeholders in the abstract syntax tree. This can be done by the user of our Eclipse plugin.

3.2 Type Placeholder

Type placeholders are used to fill out missing types in the syntax tree. They are treated like normal types during constraints generation. The unify algorithm tries to find the most general type solution for the type placeholders within the constraints set. This can either be a concrete type or a type constraint. For example the type placeholder has to be the subtype of another type. The type inference plugin later on has to fill in the unified type solutions into the inferred source code. Therefore every type placeholder keeps a reference to its parent node in the syntax tree. So the plugin is able to determine the associated statement in the source file to a given type placeholder.

3.3 Functional interfaces

In Java 8 a lambda expression is treated as an instance of a functional interface. A functional interface is an interface with only one abstract method. To type every possible lambda expression the typeless Java language needs a new set of functional interfaces called `FunN`. For all $N \in \mathbb{N}$ there is an interface

```
interface FunN<R,T1, ... , TN> {
  R apply(T1 arg1, ... , TN argN);
}
```

For each lambda expression exists a `FunN` interface which is suitable as a type. These functional interfaces play the role of function types.

3.4 Overloading

The type inference algorithm is not always able to determine a singular set of constraints out of a method call. In a typeless Java source code the only information about a method invocation is its name and the number of its parameters. In some cases multiple methods are matching with the given information. Each match implies different constraints for the types of the expressions (e.g. the given method parameters or the return type) involved in the method call. So the algorithm regards every possible case and builds several constraint sets. The cartesian product over these constraint sets represents the set of potential typings.

This means, that besides the unification, overloading is the second cause of multiple type solutions [SP15].

4 Plugin description

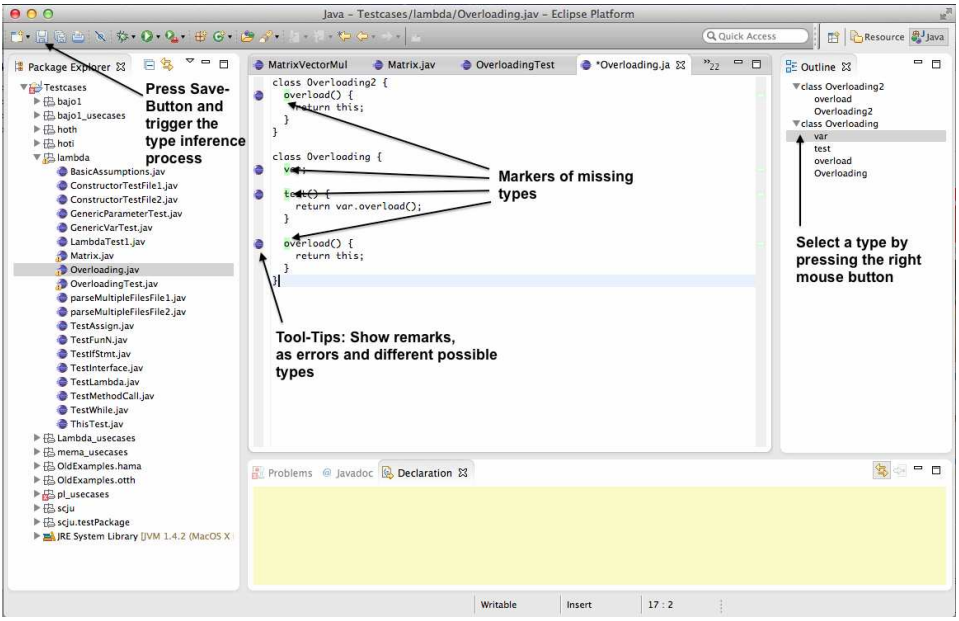

Fig. 2: Eclipse plugin

The goal of our type inference algorithm is to generate valid Java class files out of a typeless Java source code, which is achieved in three steps. First a given source file has to be inferred by the type inference algorithm. Subsequently the plugin patches the untyped statements in the source file by plugging in the inferred types. Afterwards the standard Java compiler is able to compile the resulting source code. This imposes the restriction that only valid typed Java source files are accepted by the standard Java compiler. But in most cases the type inference algorithm described in this paper generates more than one solution for the typing of a given untyped Java source code. To generate Java byte-code from the given typeless Java source code one of the infered type solutions has to be chosen.

A solution for the illustrated problem is to let the user select one of the given typings. This feature is implemented as a Eclipse plugin. In Fig. 2 the plugin is shown. The user workflow is given in Section 4.4. Section 4.1 describes the implementation, while Section 4.3 explains the user interface.

4.1 Implementation

This section describes the principles and processes of the Eclipse plugin implementation. Every time the type inference plugin is triggered on a Java document, it processes the request in the following way:

1. The source code of the document is processed by the type inference algorithm. This generates type solutions for every missing type in the syntax tree.
2. Every position in the document, where a type can be plug in gets marked by an annotation.
3. Not every type in the syntax tree is associated with a type declaration in the corresponding Java source code. Types appear in Java source code only at method declarations, field declarations and local variable declarations. Only the inferred types representing a type at the named positions are important for the Eclipse plugin. Therefore the plugin concatenates the inferred types with the previously generated markers. This is done by filtering out the types which are actually insertable into the source code and attach them to their respective markers.

4.2 Deploy type solution

The outcome of the type inference process is a list of possible type solutions.

Afterwards the plugin has to provide an interface to the user to choose one of the solutions. The implementation has to fulfill the following constraints:

Select a type for each untyped statement A source file can contain multiple statements with a missing type declaration. The user should be able to choose a type for each of those positions in the source code.

Annotations for missing types Every place a type is missing in the document an annotation is added. This annotation provides the possible types.

Integrate into user workflow Editing the document between two type selections should be possible. Every type insertion must generate a valid, although partial typeless, Java program. Therefore after each type selection the user is able to edit the source code and afterwards the other missing types can be inferred by the type inference algorithm.

Final outcome must be valid Java source code When the user selects one of the proposed types of a marker the plugin edits the document by placing in the selected type. When every marker is resolved all the missing types are patched. This process renders a valid Java source file.

There is a challenge with generic type variables. Sometimes a most general type solution contains a generic type variable. In Java a type variable must be introduced by a generic class declaration, a generic method declaration or a generic constructor declarations. Therefore a generic type variable can not be solely placed into the source code. Additionally, the declaration of the type variable has to be placed in the corresponding method or class declaration.

The source code given in Figure 3 is an example for the described practice. The types of the method `ident` and its parameter `m` are undefined. The method should be applicable to arguments of all types and the type of the parameter `m` has to extend the return type of the method `ident`. If an user triggers the plugin to inject one of the two missing types an additional declaration of a generic type variable is added. To plug in all missing types two additional type variables must be introduced (see Figure 4).

```
class GenericExample {
    ident( m ){
        return m;
    }
}
```

Fig. 3: Example: Type insertion with generics

```
class GenericExample {
    <A, B extends A> A ident( B m ){
         return m;
    }
}
```

Fig. 4: Example 3 after the plugin set in all missing types

4.3 User interface

Eclipse is a modular environment. Nearly every part of the application is replaceable or extensible through plugins. Therefore a plugin can introduce a new file type (*.jav*-file) for typeless Java code editing.

Our plugin installs only minimalistic modifications to the look and feel of Eclipse. No buttons or menu items are added. Only a new editor is used for the *jav*-files. This is needed to show different error markers and syntax highlighting than the Java editor provided by Eclipse. The editor for *jav*-files acts like the standard Eclipse Java editor.

The following controls have been added to Eclipse user interface.

Markers and annotations Our plugin displays a marker for every missing type in the *jav*-editor. These markers provide a *Quick fix* which offers the inferred types as solutions to the user.

Outline view The outline view shows an additional overview of the code displayed by the *jav*-editor.

4.4 User workflow

This section describes the workflow when using our type inference plugin for Eclipse on the basis of the example shown in Figure 5. The example source code shows a source file which defines two classes. None of the types for the methods and fields of these classes are specified. The process of inserting the missing types into the source code is running as follows (cp. Figure 2).

By saving a document in Eclipse the type inference plugin is triggered. Afterwards markers are shown in the document view which signal the possibility for a type insertion. For the field variable `var` and the method `test` in the class `Overloading` two types are possible. The annotations at this points in the source code provide a context menu with the two possible type options: `Overloading` and `Overloading2`. If the user selects one of the proposed types the plugin places the type into the source code of the document. This is done via the marker annotations displayed inside the *jav*-editor or the outline view (cp. Figure 2). After a missing type is patched the number of possible type solutions decreases. After the type of either the method `test` or the field `var` has been determined only one primary type solution remains. Figure 6 shows the outcome after selecting the type `Overloading2` for the field `var`. With only one possible type solution remaining the source code is ready for compilation. The user need not insert those missing types. Afterwards Java class files can be generated.

```
class Overloading2 {
  overload(){
    return this;
  }
}

class Overloading {
  var;

  test(){
    return var.overload();
  }
  overload(){
    return this;
  }
}
```

Fig. 5: Example for Overloading

```
class Overloading2{
  overload(){
    return this;
  }
}

class Overloading {
  Overloading2 var;

  test(){
    return var.overload();
  }
  overload(){
    return this;
  }
}
```

Fig. 6: Example for Overloading (Figure 5) after the type for `var` has been set in

5 Summary and future work

In well-known object-oriented languages as original Java or Scala, type inference is only considered in a local framework. The reason is that there was no practical implementation for a global approach. This is caused by the fact that there are often multiple solutions and it is difficult to define a principal type. This paper presents a global approach for type inference in Java 8. First it introduced a set of functional interfaces `Fun`*N*, that represent the function types. Then it described the algorithm, which determines all most general solutions. Finally an Eclipse plugin is presented, which allows the user to write a Java program without type definitions. The Eclipse plugin determines the solutions for all left out types in the Java program. If there is more than one solution the position is marked and the user can select the favored type. The plugin supports Java programming significantly, as no functional interfaces must be declared for the typing of lambda expressions, complex types with generics and wildcards need not to be detected and as the inferred types are most general the code is best possible reuseable.

5.1 Future work

Currently our type inference algorithm mostly produces multiple most general types for the same field or method when used on common Java source code. As long as no single principal type is inferred the user has to select one of the proposed type solutions. In [Pl08] we described a resolving strategy which results in an intersection type, that is principal. In future we will improve our system, such that the inferred type is the prinicipal intersection type. Additionally, the system will generate byte-code for the fields and methods typed by principal intersection types, which means for the user that the type selection would no longer be necessary.

References

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.

[La10] Lambda. Project Lambda: Java Language Specification draft, 2010. Version 0.1.5.

[Od14] Martin Odersky. *The Scala Language Specification Version 2.9*, May 2014.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *POPL 2001 Proc. 28th ACM Symposium on Principles of Programming Languages*, 36(3):41–53, 2001.

[Pl07] Martin Plümicke. Typeless Programming in Java 5.0 with Wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

[Pl08] Martin Plümicke. Intersection Types in Java. In Luís Veiga, Vasco Amaral, Nigel Horspool, and Giacomo Cabri, editors, *6th International Conference on Principles and Practices of Programming in Java*, volume 347 of *ACM International Conference Proceeding Series*, pages 181–188, September 2008.

[Pl09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

[Pl11] Martin Plümicke. Well-typings for Java$_\lambda$. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 91–100, New York, NY, USA, 2011. ACM.

[Pl14] Martin Plümicke. More type inference in Java 8. In *Perspectives of Systems Informatics - 9th International Andrei Ershov Memorial Conference, PSI 2014, St. Petersburg, Russia*, pages 168–174, 2014. Preliminary Proceedings.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[SP15] Andreas Stadelmeier and Martin Plümicke. Adding overloading to Java type inference. In *Tagungsband der Arbeitstagung Programmiersprachen (ATPS 2015)*, volume Vol-1337, pages 127–132. CEUR Workshop Proceedings (CEUR-WS.org), 2015.

[Xt15] Xtend-language, april 2015. http://www.eclipse.org/xtend/.