Java Extension: Automatic Type Inference

CS698Y Project, 2013-14 II

Prof. Rajeev Kumar {rajv@iitk.ac.in}

Abhimanyu Jaju {10327009, abhijaju@iitk.ac.in}
Harshit Maheshwari {10327290, harshitm@iitk.ac.in}
Vinit Kataria { 10327807, vinitk@iitk.ac.in}

Indian Institute of Technology, Kanpur
Computer Science and Engineering

CONTENTS

1 Abstract **3**

2 Keywords **3**

3 State of Art **3**

4 History and Usefulness of auto **4**

4.1 History 4

4.2 Usefulness 4

4.2.1 Java Bugs 5

5 C++11 Specifications **6**

6 C++14 Proposed Plan **7**

7 Proposed Rules **9**

7.1 'auto' for variables 9

7.1.1 'auto' for primitive data types 9

7.1.2 'auto' for user defined objects 10

7.2 'auto' for functions 11

7.2.1 Multiple return types for primitives 11

7.2.2 Multiple return types for classes 12

7.3 C/C++ Compiler 13

7.3.1 Recursion 14

7.4 Java Compiler: Possibilities 15

8 limitation **17**

9 Implementation **18**

1 Abstract

In Java, the type of a variable must be explicitly specified in order to use it. However, with our knowledge of type-inference and type-unification, usually we can deduce the types of variables as well as return types of functions (although it is not always possible to deduce the type). For this purpose we propose the use of auto keyword in Java. This would help developers to focus on the logic rather than on things which the compiler can itself deduce. The feature of auto keyword for type-deduction of variables has already been included in the latest C++11 standard. Further, in the proposed C++14 standard, automatic deduction of function-return-type has been included.

2 Keywords

Java language features, auto, function type deduction, variable type deduction

3 State of Art

Currently, Java supports no such feature. In Java, the type of the variable and the return type of functions have to be explicitly mentioned at the time of variable/function declaration. However, the C++ language specifications supports leaving the type deduction up to the compiler whenever possible in certain situations. The C++11 standard supports the keyword **auto** for variables which allows automatic type deduction for variables at compile time. Hence, at the time of variable declaration, the compiler deduces the type of the variable by looking at the value being assigned to it. Further, the new C++14 standard extends the usage of the keyword auto to function (as well as for lambda function) return-type-deduction. On similar lines, C# standard currently supports the var keyword for automatic type deduction of variables. But, a necessity in both of the implementations is that the variable has to be initialized at the time of declaration. Another necessity is that if the function return type deduction has to be done by the compiler, then the function definition needs to accompany the function declaration.

4 History and Usefulness of auto

The auto specifier [1, 2] was only allowed for variables declared at block scope or in function parameter lists. It indicated automatic storage duration, which is the default for these kinds of declarations. The meaning of this keyword was changed in C++11.

4.1 History

"The auto feature has the distinction to be the earliest to be suggested and implemented: I had it working in my Cfront implementation in early 1984, but was forced to take it out because of C compatibility problems. Those compatibility problems disappeared when C++98 and C99 accepted the removal of "implicit int"; that is, both languages require every variable and function to be defined with an explicit type. The old meaning of auto ("this is a local variable") is now illegal. Several committee members trawled through millions of lines of code finding only a handful of uses – and most of those were in test suites or appeared to be bugs.
Being primarily a facility to simplify notation in code, auto does not affect the standard library specification."

– Bjarne Stroustrup , *C++11 - the new ISO C++ standard [**?**]*

4.2 Usefulness

As stated by Bjarne Stroustrup, auto is basically a tool to simplify notation in the code and at the same time not effect the standard library specification. As a result, use of auto in extensions of older code would not lead to broken code. Some powerful use of **auto** [9, 10] are described below:

- **Readability:** According to Java Language Specification (JLS) type of a variable has to be explicitly mentioned at the time of declaration. Sometimes, in case of castings this can become redundant and messy leading to reduced readability. For example, to iterate over a map:

```
Map map = getSomeMap();
    Iterator iter = map.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry entry = (Map.Entry) iter.next();
        MyTableValue value = (MyTableValue) entry.
            getValue();
        ...;
    }
```

Iteration in java

Here the variable declarations of 'entry' and 'value' need to specify the type twice. A simple extension to the JLS would be to be able to introduce variables without specifying the type.

```
auto map = getSomeMap();
    auto iter = map.entrySet().iterator();
    while (iter.hasNext()) {
        auto entry = (Map.Entry) iter.next();
        auto value = (MyTableValue) entry.getValue();
        ...;
    }
```

auto as iterator

This makes the code easier to read, type, and change due to less duplication of information.

- **Maintainability:** If we use ‘auto‘ in source code of large projects, then changes to source code can be made easily if we had used ’auto’ previously.

- **Ease of use:** The use of auto to deduce the type of a variable from its initializer is obviously most useful when that type is either hard to know exactly or hard to write.

```
void printall(com.google.android.maps.Map myMap)
{
  ...
  //instead of com.google.android.maps.Map temp  = myMap
  auto temp = myMap;
  ...
}
```

Type is hard to type/know

4.2.1 Java Bugs

There are some attempts [4, 7] (bug 4459053 and 6242254) in which auto type inferencing was tried to be introduced in Java, which were unsuccessful. The reason given against this was primarily the one we give for it’s use: *readability*.
The evaluation committee gave the following reasons for not including automatic type inferencing:

- The redundant type serve as valuable documentation. Readers do not have to search for declaration to find the actual type of variable.

- The redundancy allows the programmer to declare the intended type, and thereby benefit from a cross check performed by the compiler, that helps catch errors.

Several other related bugs were filed by users but were repeatedly rejected by the evalutaion committee [2, 5, 6, 8].

5 C++11 Specifications

The C++11 specification [3] for the use of ‘auto‘ keyword list the rules as follows:

- The ‘auto‘ keyword can be used as a simple type specifier. Examples:

```
int foo();
auto x1 = foo(); // x1 : int
const auto& x2 = foo(); // x2 : const int&
auto& x3 = foo(); // x3 : int&: error, cannot bind a
   reference to a temporary
float& bar();
auto y1 = bar(); // y1 : float
const auto& y2 = bar(); // y2 : const float
auto& y3 = bar(); // y3 : float
A* fii()
auto* z1 = fii(); // z1 : A*
auto z2 = fii(); // z2 : A*
auto* z3 = bar(); // error, bar does not return a pointer
    type
```

Example assignment using ‘auto‘

- ‘auto‘ can be used to provide a effective way for the programmers to express his intentions in context of objects. Examples:

```
A foo();
A& bar();
...
A x1 = foo(); // x1 : A
auto x1 = foo(); // x1 : A
A& x2 = foo(); // error, we cannot bind a non-lvalue to a
    non-const reference
auto& x2 = foo(); // error
A y1 = bar(); // y1 : A
auto y1 = bar(); // y1 : A
A& y2 = bar(); // y2 : A&
auto& y2 = bar(); // y2 : A&
```

Reference type assignments using ‘auto‘

- In C more than one variable can be declared in a single assignment provided that individual type deductions don’t leave conflicts.

```
int i;
auto a = 1, *b = &i;   //ok
auto x = 1, *y = &x;   //Valid assignment from left to
   right
```

```
auto c = 1, d=2.2;     // Error type conflicts c:int; d:
   double;
```

Multi Variable Declaration

- ‘auto‘ can be used for direct initialization, for the purpose of type deduction. Example:

```
auto x = 1; // x : int
auto x(1); // x : int
auto* x = new auto(1);   // x : int *
```

Multi Variable Declaration

6 C++14 Proposed Plan

Some of the proposals for C++14 [9] language specification with reference to ‘auto‘ are mentioned below.

- Allowing non-defining function declarations with auto return type is not strictly necessary, but it is useful for coding styles that prefer to define member functions outside the class. Example:

```
struct A {
  auto f(); // forward declaration
};
auto A::f() { return 42; }
```

Forward declaration

- Since C++ compilers are single parse, if the return type cannot be deduced from the first return statement then it gives error.

```
A foo();
A& bar();
...
A x1 = foo(); // x1 : A
auto x1 = foo(); // x1 : A
A& x2 = foo(); // error, we cannot bind a non-lvalue to a
    non-const reference
auto& x2 = foo(); // error
A y1 = bar(); // y1 : A
auto y1 = bar(); // y1 : A
A& y2 = bar(); // y2 : A&
auto& y2 = bar(); // y2 : A&
```

Function return type deduction

- Similarly, for templates, some examples:

```
A foo();
A& bar();
...
A x1 = foo(); // x1 : A
auto x1 = foo(); // x1 : A
A& x2 = foo(); // error, we cannot bind a non-lvalue to a
    non-const reference
auto& x2 = foo(); // error
A y1 = bar(); // y1 : A
auto y1 = bar(); // y1 : A
A& y2 = bar(); // y2 : A&
auto& y2 = bar(); // y2 : A&
```

Template forward declaration

- Type deduction for multiple returns in a function is also defined. Examples:

```
auto iterate(int len)
{
  for (int i = 0; i < len; ++i)
    if (search (i))
      return i;
  return -1;
}
```

Multiple returns in a function

- Recursion is handled in the following manner:

```
auto h() { return h(); } // error, return type of h is
   unknown

auto sum(int i) {
  if (i == 1)
    return i;          // return type deduced to int
  else
    return sum(i-1)+i; // ok to call it now
}
```

Type deduction in recursive functions

Table 7.1: A comparative study

Feature	**C++11/C++14**	**Proposed Rule in Java**
Variable initialization (primitive type)	[C++11]:Type is assigned according to the C++11 ranges for primitive data types	Type is assigned according to Table 7.2
Variable initialization (user defined class)	[C++11]:Type is assigned according to type of value being assigned	Type is assigned according to type of value begin assigned
Multiple variable initialization in a single statement	[C++11]:Allowed, but type should be same for all	Allowed. Also, type can be different
Function Return Type	[C++14]: Type should be inferable by the first return statement	Type should be inferable through at least one return statement/cyclic dependency
Multiple Return Type	[C++14]: Type must be same	Type can be different. Least common ancestor in inheritance hierarchy is returned.

7 Proposed Rules

In Table 7.1 we do a comparative analysis of auto usage in different C++11/14 and Java.

7.1 'auto' for variables

These rules discuss 'auto' type assignment w.r.t. variables.

```
auto x = VALUE_TO_BE_ASSIGNED //right hand side can be
   expression also
```

Listing 1: Example assignment for variables

7.1.1 'auto' for primitive data types

If auto is used for variables then for primitive types we propose the type assignments based on the range of the value to be assigned as mentioned in Table 7.2. However, as per the language specifications if 'l' is appended in the numeral literal it is considered as a long literal by default. Similarly, if 'f' is appended in the decimal literal it is considered as a float literal by default.

Table 7.2: Range for ‘type‘ assignment

Primitive Type	**Lower Range**	**Upper Range**
int	-2,147,483,648	2,147,483,647
long	(-9,223,372,036,854,775,808 ... -2,147,483,649)	(2,147,483,648 ... 9,223,372,036,854,775,807)
float	1.4E-45	3.4028235E+38
double	439E-324	1.7976931348623157E+308
boolean	true	false

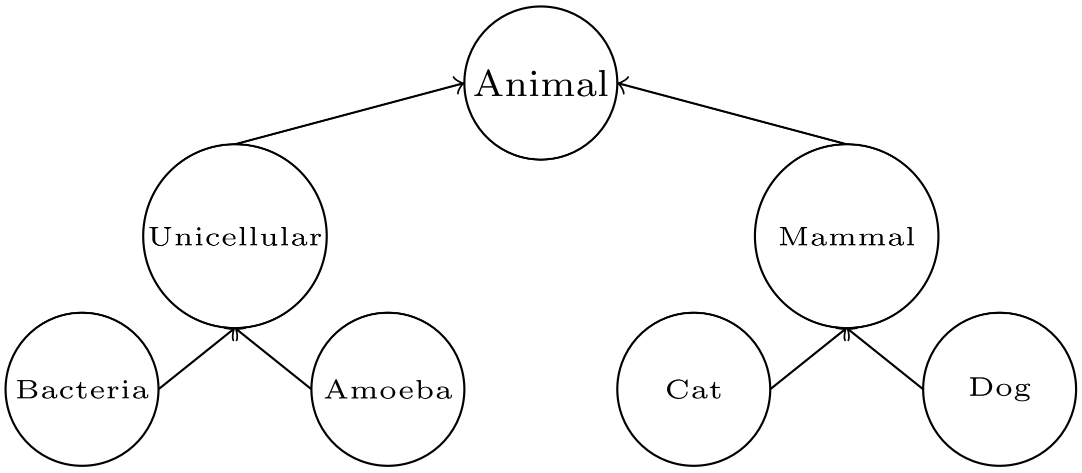

Figure 7.1: Class Hierarchy Diagram

7.1.2 ‘AUTO‘ FOR USER DEFINED OBJECTS

While assigning user defined objects to auto variables the type of the object being assigned is given to the auto variable. Suppose we have the following Class arrangement as shown in Figure 7.1.

```
auto x = new Animal();     // x is assigned type 'Animal'
auto y = (Dog) animal;   // x is assigned type 'Dog'
```

Listing 2: Example assignment of user defined objects

7.2 ‘AUTO‘ FOR FUNCTIONS

In functions instead of specifying return type in function signature we can use auto as return type. The compiler will infer type from all return statements of the function and will try to deduce the most specialized return type which will be compatible with all return statements. This will allow users to directly return primitive types/objects at any level of hierarchy. Specific rules are explained through the examples below. In the following code sample ‘x‘ would be assigned type ‘int‘ and ‘y‘ would be assigned type ‘Animal‘.

```
auto myFunct1(){
  ...
  return 4;
}

auto myFunct2(){
  ...
  return new Animal();
}

auto x = myFunct1();    //x: int
auto y = myFunct2();    //y: Animal
```

Listing 3: Basic function calling

7.2.1 MULTIPLE RETURN TYPES FOR PRIMITIVES

In case of conflicting return types of primitive data the function will return lowest common ancestor as shown in Figure 7.2. Same rule will be followed for the wrapper class of these return types.

```
auto myFunct(){   //return type: double
  int i;
  double d;
  ...
  if(condition){
    return i;
  }
  else{
    return d;
  }
  ...
}
```

Listing 4: Multiple return types

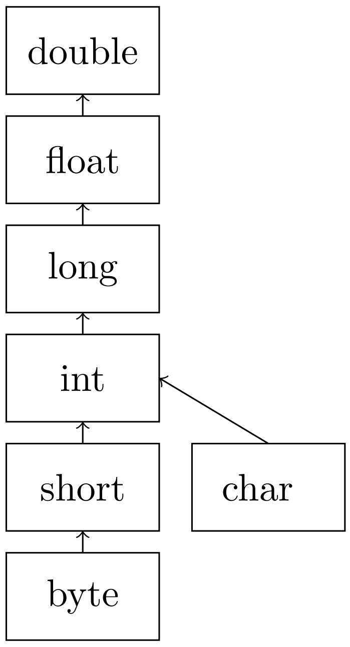

Figure 7.2: Primitive data types coercion

7.2.2 Multiple return types for classes

Parent ↔ Child
When the return type of a function is auto and it returns parent class as well as child class then after type resolution the parent class would be assigned as the return type of the function.
Based on the hierarchy of Figure 7.3 if we have the code as given in Code 5 then the return type should be Animal.

```
auto myFunct(){        //return type: animal
  Animal a = new Animal();
  Mammal m = new Mammal();
  ...
  if(condition){
    return a;
  }
  else{
    return m;
  }
  ...
}
```

Listing 5: Multiple return types

Different levels in inheritance hierarchy

In case when the return type of the function is auto and it returns two sibling class in a class hierarchy then the return type of the function would be the lowest common ancestor in the inheritance hierarchy.

Based on the hierarchy of 7.1 if we have the code as given in Code 6 then the return type should be Animal as it is the lowest common ancestor in the class hierarchy.

```
auto myFunct(){          //return type: animal
  Amoeba a = new Amoeba();
  Cat m = new Cat();
  ...
  if(condition){
    return a;
  }
  else{
    return c;
  }
  ...
}
```

Listing 6: Multiple return types

This is allowed because currently in Java the code given in Code 7 is allowed.

```
Animal myFunct(){          //this is allowed in java
  Amoeba a = new Amoeba();
  Cat m = new Cat();
  ...
  if(condition){
    return a;
  }
  else{
    return c;
  }
  ...
}
```

Listing 7: Multiple return types

7.3 C/C++ Compiler

In C++11 standards value assignment to ‘auto‘ variables cannot be deferred and the variable definition should be accompanied together with variable declaration. Code given in Code 8 is allowed but code given in Code 9 gives error.

```
auto x = 11;  //declarationa and definition should be
    together
```

Listing 8: Immediate variable definition

```
auto x;   //gives error; definition should accompany
    declaration
...
...
x=11;
```

Listing 9: Deferred variable definition

C/C++ compilers are single parse compilers. Therefore, we need to give the function definition/declaration before actual function use.

7.3.1 Recursion

The code given in Code 10 and Code 11 should give error while the code given in Code 12 is allowed. The reason is that we should be able to deduce the return type of functions in the first parse as C/C++ compilers are single parse.

```
auto h() {
  return h();      //gives error
}
```

Listing 10: Not allowed

```
auto sum(int i) {
  if (i == 1)
    return sum(i-1)+i;
  else
    return i;
}
```

Listing 11: Not allowed

```
auto sum(int i) {
  if (i == 1)
    return i;
  else
    return sum(i-1)+i;
}
```

Listing 12: Recursion allowed

7.4 Java Compiler: Possibilities

We can allow the code given in Listing 15 in Java as Java compilers make multiple parse over the code. This is also the reason that in Java we can defer the function definition after function use because we can parse the code again to type check with the function definition.

```
auto sum(int i) { //allowed
  if (i == 1)
    return sum(i-1)+i;
  else
    return i;
}
```

Listing 13: Deferred variable definition

In fact, in Java we can allow the use of ‘auto‘ keyword for function return type for cyclic dependencies as shown in Code 14. In Code 14 the return type of all the functions will become ‘int‘.

```
auto myFunct1(){       //return type: int
  ...
  return myFunct2();
}

auto myFunct2(){       //return type: int
  ...
  return myFunct3();
}

auto myFunct3(){       //return type: int
  int i;
  ...
  if(condition){
    return myFunct1();
  }
  else{
    return i;
  }
}
```

Listing 14: Cyclic Dependency

There is only one condition that it is actually possible to deduce the return type and there is not clash in return types. For instance the code given in Listing 15 will give compile time error as there is unresolved cyclic dependency of return types.

```
auto myFunct1(){    //compilation error: return type cannot
    be deduced
  ...
  return myFunct2();
}

auto myFunct2(){
  ...
  return myFunct3();
}

auto myFunct3(){
  ...
  return myFunct1();
}
```

Listing 15: Type deduction not possible

However, the code given in Code 16 is allowed and we can deduce the return type.

```
auto myFunct1(){    //return type animal
  ...
  return myFunct2();
}

auto myFunct2(){    //return type animal
  Amoeba a = new Amoeba();
  ...
  if(condition){
    return myFunct3();
  }
  else{
    return a;
  }
}

auto myFunct3(){    //return type animal
  Cat c = new Cat();
  ...
  if(condition){
    return myFunct1();
  }
  else{
    return c;
  }
}
```

Listing 16: Cyclic dependencies

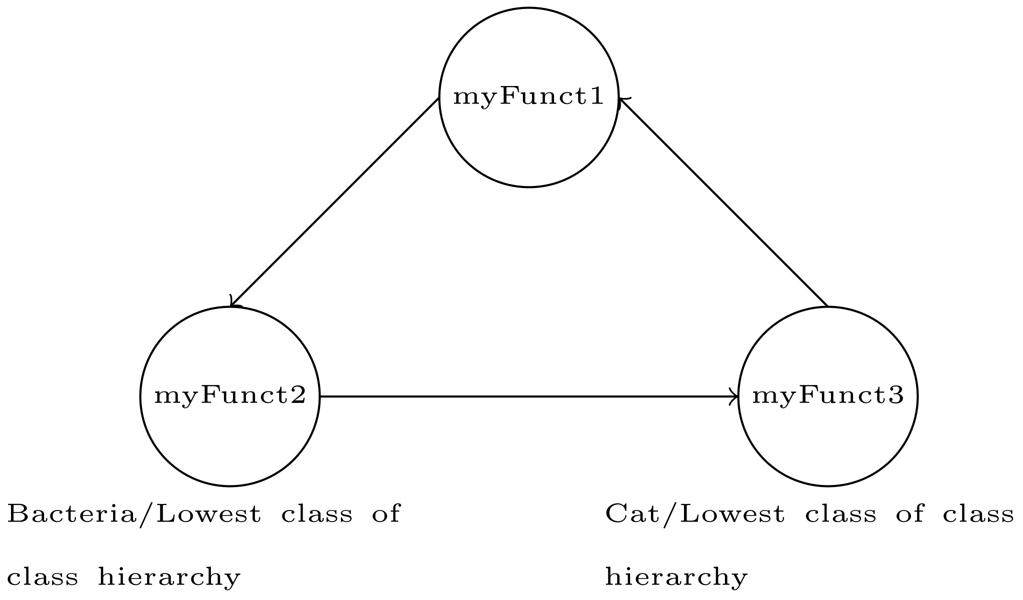
myFunct1
myFunct2
myFunct3
Bacteria/Lowest class of class hierarchy
Cat/Lowest class of class hierarchy

Figure 7.3: Cyclic dependencies

8 LIMITATION

The following are some limitations of ‘auto‘:

- auto as type in parameter(s) of functions
  The function calls are made during run-time so that the data type of arguments cannot be found entirely during compile time. So we have a run-time dependency in determining the actual type of auto parameters used in the function definition.

- Overestimating the power of auto
  Allowing rules like type inferencing in cyclic dependencies will put the responsibility of ensuring that return type can be inferred on developers. This might result in unnecessary confusion and longer time spent in debugging.

- Loss of comprehensibility
  Extensive use of auto can lead to problems in understanding and readability of code.

- Also, redundant type checking is useful for cross checking which leads for trusted and safer code.

9 Implementation

We have modified the Java grammar and made the following changes:

```
AutoType:
  AUTO
;

FieldDeclaration:
    AutoType AutoVariableDeclarators SEMICOLON
|   Modifiers AutoType AutoVariableDeclarators SEMICOLON
|   ...
;

AutoVariableDeclarators:
  AutoVariableDeclarator
| AutoVariableDeclarators COMMA AutoVariableDeclarator
;

AutoVariableDeclarator:
  VariableDeclaratorId EQUALS VariableInitializer
;


MethodHeader:
  AutoType MethodDeclarator
| Modifiers AutoType MethodDeclarator
| ...
;

LocalVariableDeclaration:
    AutoType AutoVariableDeclarators
| ...
;
```

Listing 17: Modified Grammar

We have implemented and rigourously tested the following:

- We have implemented a basic functionality providing Java compiler as the starting point for our project which supports compile time type deduction for variable declarations and return type of functions. We also generate assembly code for the same.
- We are doing type deductions only at compile time.
- **auto** can be used for type deductions of primitive as well as user defined types.

```
auto x=3, y = 'a';
class myClass{
  ...
}
auto z =  new myClass();
```

```
x:int   y:char



z: myClass
```

- We have also generated assembly code for the same.
- We have also implemented type deductions for primitive function return type.

```
auato myFunction(){
  int i = 0;
  return i;
}

auto myFunction(){
  int i = 1;
  int d = 2.2;
  if (condition){
    return i;
  }
  else{
    reutn d;
  }
}
```

```
return type: int
  i: int



return type: double
  i: int
  d: double

  returning: int


  returning: double

.
```

- Automatic type coersion is also implemented for the same.
- Type deductions for **auto** functions with only one return statement which returns object is implemented.

```
auto myFunction(){
  Animal a;
  Animal b;
  if(condition){
    return a;
  }
  else{
    return b;
  }
}
```

```
return type: Animal
  a: Animal
  b: Animal

  returning: Animal


  returning: Animal

.
```

- Return type deduction for **auto** functions with multiple return statements that are returning different/same objects is also implemented.

```
auto myFunction(){
  Animal a;
  Dog d;
  if(condition){
    return a;
  }
  else{
    return d;
  }
}
```

```
return type: Animal
  a: Animal
  d: Dog

  returning: Animal


  returning: Dog


.
```

The following is not implemented in out implmentation:

- Return type inferencing for cyclic dependencies in functions is not implemented due to the immense complexity.

REFERENCES

[1] Auto - a necessary evil? http://www.howzatt.demon.co.uk/articles/AutoP1.html.

[2] C++11. http://en.cppreference.com/w/cpp/language/storage_duration.

[3] C++11. https://en.wikipedia.org/wiki/C++11/.

[4] Jdk-4459053 : Type inference for variable declarations. http://bugs.java.com/view_bug.do?bug_id=4459053, May 2001.

[5] Jdk-4879776 : Constructor type inference (jsr14 + jsr65 ++). http://bugs.java.com/view_bug.do?bug_id=4879776, 2003.

[6] Jdk-6220689 : Type arguments for a class shall be inferred on constructor invocation. http://bugs.java.com/view_bug.do?bug_id=6220689, 2005.

[7] Jdk-6242254 : Language support for type inference. http://bugs.java.com/view_bug.do?bug_id=6242254, 2005.

[8] Jdk-6840638 : Project coin: Improved type inference for generic instance creation (aka 'diamond'). http://bugs.java.com/view_bug.do?bug_id=6840638, May 2009.

[9] Gabriel Dos Reis Bjarne Stroustrup, Jaakko JÃď'rvi. Deducing the type of variable from its initializer expression. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1984.pdf, 2006.

[10] Bjarne Stroustrup. C++11 - the new iso c++ standard. `http://www.stroustrup.com/C++11FAQ.html#auto`.