Java Extension: Automatic Type Inference

CS698Y Project, 2013-14 II

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

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 have 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 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

Some powerful use of **auto** are described below:

- **auto** can be used for iterating through object lists.

```
void f(vector<double>& v)
{
  for (auto x : v) cout << x << '\n';
  for (auto& x : v) ++x;  // using a reference to allow
     us to change the value
  for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x
     << '\n';
}
```

Listing 1: auto as iterator

- 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.

```
template<class T> void printall(const vector<T>& v)
{
  for (auto p = v.begin(); p!=v.end(); ++p)
    cout << *p << "\n";
}
```

Listing 2: Type is hard to type/know

5 C++11 SPECIFICATIONS

The C++11 specification 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 : ïňĆoat
const auto& y2 = bar(); // y2 : const ïňĆoat&
auto& y3 = bar(); // y3 : ïňĆoat&
A* fii()
auto* z1 = fii(); // z1 : A*
auto z2 = fii(); // z2 : A*
auto* z3 = bar(); // error, bar does not return a pointer
    type
```

Listing 3: 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 nonâĹŠlvalue to
    a nonâĹŠconst 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&
```

Listing 4: 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;
```

Listing 5: 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 *
```

Listing 6: Multi Variable Declaration

6 C++14 Proposed Plan

Some of the proposals for C++14 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; }
```

Listing 7: Forward declaration

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

```
auto Correct(int i) {
  if (i == 1)
    return i;               // return type deduced as int
  else
    return Correct(i-1)+i;  // ok to call it now
}
auto Wrong(int i){
  if(i != 1)
    return Wrong(i-1)+i;  // Too soon to call this. No
       prior return statement.
  else
    return i;             // return type deduced as int
}
```

Listing 8: Function return type deduction

- Similarly, for templates, some examples:

```
auto Correct(int i) {
  if (i == 1)
    return i;               // return type deduced as int
  else
    return Correct(i-1)+i;  // ok to call it now
}
auto Wrong(int i){
  if(i != 1)
    return Wrong(i-1)+i;  // Too soon to call this. No
       prior return statement.
  else
    return i;             // return type deduced as int
}
```

Listing 9: 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;
}
```

Listing 10: 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
}
```

Listing 11: Type deduction in recursive functions

7 Proposed Rules

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 12: 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 following type assignments based on the range of the value to be assigned: However, as per the

Table 7.1: 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

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.

7.1.2 'auto' for user defined objects

Suppose we have the following Class arrangement as shown in Figure 8.1.

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

Listing 13: Example assignment of user defined objects

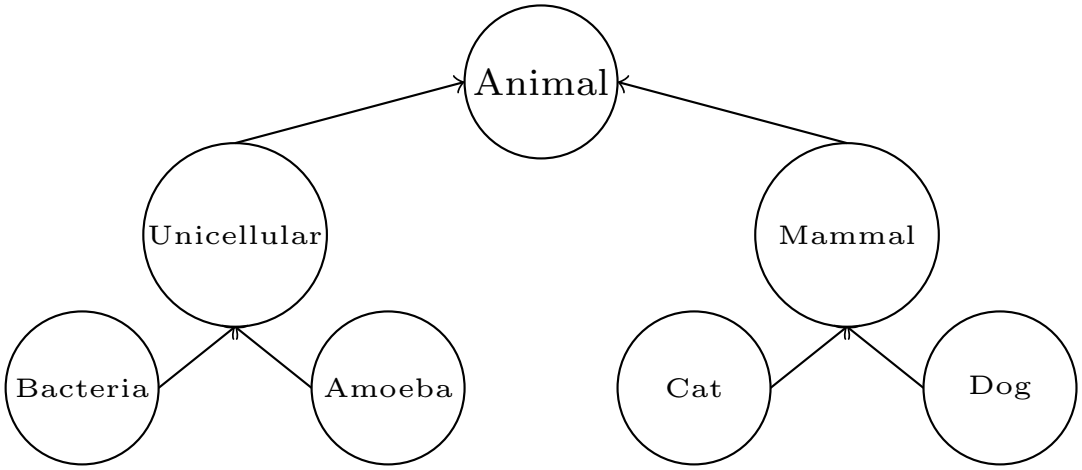

Figure 7.1: Class Hierarchy Diagram

7.2 'AUTO' FOR FUNCTIONS

In the following code sample 'x' would be assigned type 'int' and 'y' would be assigned type 'Animal'.

```
int myFunct1(){
...
}

Animal myFunct2(){
...
}

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

Listing 14: Basic function calling

7.2.1 MULTIPLE RETURN TYPES FOR PRIMITIVES

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

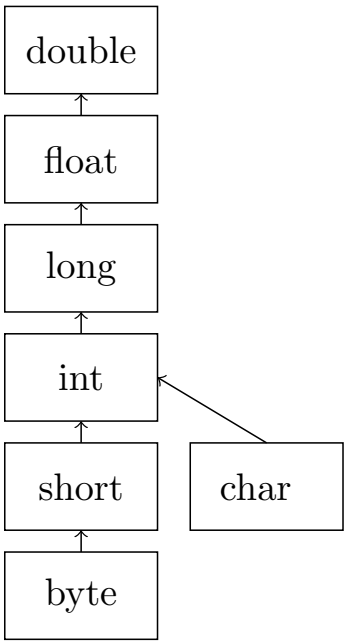

Figure 7.2: Primitive data types coercision

```
    else{
      return d;
    }
    ...
}
```

Listing 15: Multiple return types

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

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 8.1 if we have the code as given in Listing 7 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 16: Multiple return types

Sibling ↔ Sibling

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 Figure 8.1 if we have the code as given in Listing 13 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 17: Multiple return types

This is allowed because currently in Java the code given in Listing 13 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 18: 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 Listing 3 is allowed but code given in Listing 5 gives error.

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

Listing 19: Immediate variable definition

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

Listing 20: 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 Listing 4 and Listing 7 should give error while the code given in Listing 7 is allowed. The reason is that we should know be able to deduce the return type of functions in the first parse as C/C++ compilers are single parse.

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

Listing 21: Not allowed

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

Listing 22: Not allowed

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

Listing 23: Recursion allowed

7.4 JAVA COMPILER

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 24: 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 Listing 21. In Listing 21 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 25: 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 26: Type deduction not possible

However, the code given in Listing 27 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();
  ...
```

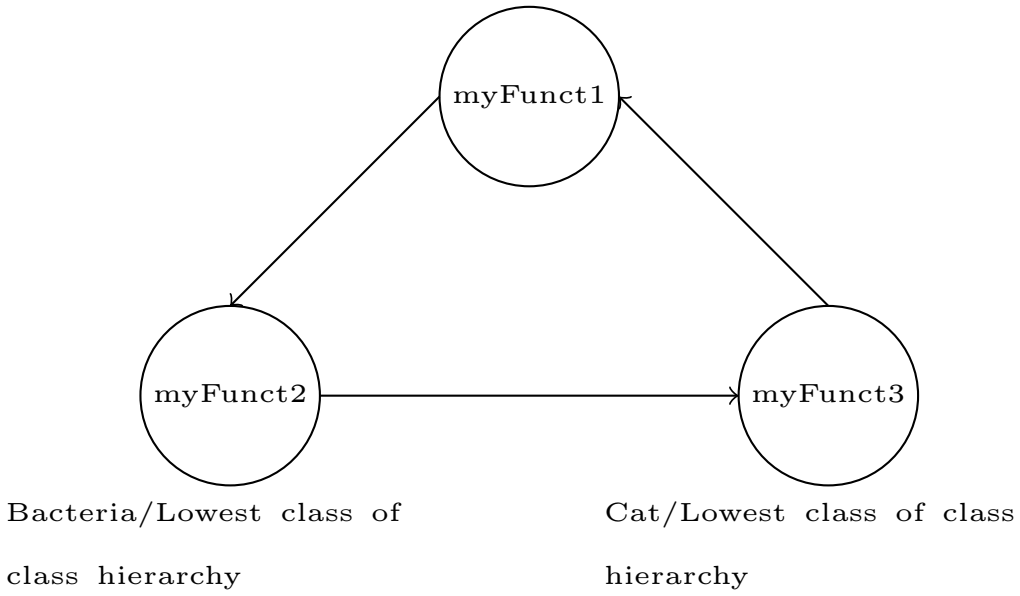

Figure 7.3: Analysis of Listing 18

```
  if(condition){
    return myFunct3();
  }
  else{
    return a;
  }
}

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

Listing 27: Cyclic dependencies

8 IMPLEMENTATION

As of now, we have implemented 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';  //x:int   y:char

class myClass{
  ...
}
auto x =  new myClass();  //x: myClass
```

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

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

auto myFunction(){    //return type: double AFTER type
   coercsion
  int i = 1;
  int d = 2.2;
  if (condition){
    return i;   //return: int
  }
  else{
    reutn d;    //return: 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(){     //return type is animal
  Animal a;
  Animal b;
  if(condition){
    return a;
  }
  else{
    return b;
  }
}
```

The following is yet to be implemented:

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

```
auto myFunction(){     //return type should be Animal
  Animal a;
  Dog d;
  if(condition){
    return a;    //return type is animal
  }
  else{
    return d;    //return type is dog
  }
}
```

- Return type inferencing for cyclic dependencies in functions is yet to implemented.
- Run time type deduction cases are yet to be implemented.

9 ACKNOWLEDGEMENTS