Fakultät für Informatik

Technische Universität München

Master seminar

C++0x Type Inference

Stefan Schulze Frielinghaus
November 18, 2009

Abstract

This article presents the new type inference features of the upcoming C++ standard. The main focus is on the keywords *auto* and *decltype* which introduce basic type inference. Their power but also their limitations are discussed in relation to full type inference of most functional programming languages.

1 Introduction

Standard C++ requires explicitly a type for each variable (s. [6, § 7.1.5 clause 2]). There is no exception and especially when types get verbose it is error prone to write them. The upcoming C++ standard, from now on referenced as C++0x, includes a new functionality called *type inference* to circumvent this problem. Consider the following example:

```
int a = 2;
int b = 40;
auto c = a + b;
```

In this case the compiler may figure out the type of variable '`c`' because the type of the rvalue of expression '`a + b`' is known at compile time. In the end this means that the compiler may look up all variables and function calls of interest to figure out which type an object should have. This is done via type inference and the *auto* keyword. Often types get verbose when templates are used. These cases demonstrate how useful the *auto* keyword is. For example, in C++98 and 03 developers often use *typedef* to circumvent the problem of verbose types:

```
typedef std::vector<std::string> svec;
...
svec v = foo();
```

In C++0x a *typedef* may be left out in favor of *auto*:

```
auto v = foo();
```

In some cases it may reduce the overhead if someone is not familiar with a code and would therefore need to lookup all *typedefs* (s. appendix A for a complete example). With *auto* this kind of verbosity can be suppressed.

A similar function is introduced by the *decltype* keyword where the type of an object is referenced explicitly by the developer and not figured by the compiler as it is done with the *auto* keyword:

```
int a = 2;
decltype(a) b = 40;
```

In this example object '`b`' should have the same type as object '`a`'. This is especially handy for function return types:

```
template <class T>
T foo (T a, T b) {
        return a * b;
}
```

If the multiplication operator is overloaded with another return type than '`T`', then a second template argument must be specified as the return type.

```
template <class T, class U>
U foo (T a, T b) {
        return a * b;
}
```

But this introduces another problem, template argument deduction does not work for return types and therefore, the template parameters must be specified explicitly. All this may be circumvented with the *decltype* keyword. Both keywords are discussed in the proposals [8], [9] and are now in the official draft [3] for the upcoming release of C++0x.

This article discusses the *auto* keyword in more detail in section 2 and shows the full power of *decltype* in section 3. Compiler support for type inference is discussed in section 4.

2 The *auto* keyword in-depth

The more verbose a type gets the more useful is a feature as the *auto* keyword. Especially when templates are instantiated the type of an object may get verbose. For example, it is common to use an iterator to go through the elements of a vector:

```
for (std::vector<std::string>::const_iterator i = v.begin();
                i != v.end(); ++i) { }
```

Via the *auto* keyword this can be changed to the following:

```
for (auto i = v.cbegin(); i != v.cend(); ++i) { }
```

The compiler checks the type of object '`v`' and figures from that the return type of function '`cbegin`'. Therefore, the type of iterator variable '`i`' can be deduced during compile time. For const-correctness new functions '`cbegin`', '`cend`', '`crbegin`' and '`crend`' are proposed in [1]. They work similar to their counterparts without '`c`' in the beginning but return const objects.

Limitations

The *auto* keyword introduces some kind of type inference but is still limited.

> "The semantics of auto should follow exactly the rules of template argument deduction." [8, p. 10]

Template argument deduction is described in full detail in [6, § 14.8.2]. A more verbose description can be found in [15, chapter 11].

Specifying the type for a template argument may lead to verbose code, is error prone to code changes and makes it hard to use template-overloading. Therefore, it is often better to let the compiler deduce the actual type for a template argument:

```
template<class T>
T mul(T a, T b) {
        return a * b;
}

mul<int>(21, 2); // 1: type specified explicitly
mul(21, 2);      // 2: type deduced by the compiler
mul(21, 2.0);    // 3: deduction fails,
                 //    no matching function mul<int,double>
```

The first step in the deduction process is to compare the types of the function call arguments. The template function '`mul`' expects two values, both of the same type. Function call 1 of the example specifies the type explicitly and since both arguments are of type `int`, template instantiation works. In function call 2 both arguments are also from the same type. Therefore, template deduction works, too. In function call 3 the type of the first and second argument differ and the template deduction fails. This does not mean that the source code is broken. If another template is left where the deduction process works, then that template will be used (s. function

template overloading [6, § 14.5.5.1] and SFINAE [15]). To accomplish a look-up for a type of a variable the compiler needs a data structure where the type can be found. In general a symbol-table is used which includes typical attributes as the name, scope, size and type of a symbol [12, p. 45]. In C++ basic types as e.g. `int`, `char` and so on are predefined. All other types are composed types and often stored as tuples. Therefore, a look-up of a type is a crucial task for type deduction and needs to be done in a fast and efficient way. The actual implementation is left to the compiler vendor and is not described in the C++ standard [6].

In the next step a check is done if all agreed resources are available for the deduced type. For the example above this means that the multiplication operator must be defined for type `int`. There exist a few exceptions where the compiler may change the types of template parameters (call them `P`) and function arguments (call them `A`) to make them compatible:

- If `A` is a cv-qualified type and `P` is not a reference type, then the top-level cv-qualifiers of `A` are omitted during the deduction process [6, § 14.8.2.1 clause 2]:

  ```
  template <class T> void foo (T t) { }
  int const x = 42;
  foo (x);
  ```

  Function '`foo`' is called with a const variable but expects a non-const variable.

- If `P` is a cv-qualified type, then the top-level cv-qualifiers of `P` are omitted during the deduction process [6, § 14.8.2.1 clause 2]:

  ```
  template <class T> void foo (T const t) { }
  int x = 24;
  foo (x);
  ```

  Function '`foo`' is called with a non-const variable but expects a const-variable.

- While cv-qualifiers may be ignored during the deduction process, they are still valid and used after the deduction is completed. That means they may be ignored to find a matching type but are still used afterwards.

- Overload resolution: The class hierarchy may be compared to find a possible template argument type [6, § 14.8.2.4 clause 8 and § 14.8.3 clause 4]:

```
template <class T> struct B { };
template <class T> struct D : public B<T> { };
template <class T> void foo (B<T> &b) { }

B<int> b;
D<int> d;
foo (b);
foo (d); // calls foo((B<int>&) d)
```

The upcast of object `d` from type `D<int>` to `B<int>&` is done automatically during the deduction process.

- Function and array types may be converted to pointer types.

```
template<class T> void foo (T t) { }
template<class T> void bar (T &t) { }

int x[42];
foo (x);
bar (x);
```

In function '`foo`' `T` converts to type `int*` while in function '`bar`' `T` converts to type `int[42]`.

Beside the few exceptions template argument deduction results in deducing the type of a function argument and substituting all template parameter identifiers.

This technique is used for type inference in C++0x, too. But *auto* is not always as powerful as a template. For example, consider the following code:

```
template <class T> void foo (T x) { }

void bar (auto x) { } // not valid
                      // since x is not initialized

int main (void) {
        foo (42);
        bar (42);
}
```

In this case function '`bar`' is ill-formed because variable '`x`' is not initialized while it is declared. The standard clearly states that a variable must be initialized via an *init-declarator* as e.g. an assignment-expression, if *auto* is used. This already indicates that *auto* is not as powerful as templates

in general. Templates are instantiated whenever e.g. a call to a template function is done. This means that one template function may expand to several specialized functions. But *auto* stands as a placeholder for exactly one type and not for a whole family of types. Most functional programming languages support full type inference, but this is not the case in C++0x and the *auto* keyword. According to [3, § 7.1.6.4] every type of an object is deduced by its initializer or is specified explicitly at the end of a function declaration. Haskell as an example of a functional programming language does not have such limitations. Thus, consider the following Haskell code:

```
foo x = x * x

foo 42
foo 42.0
```

The type of variable '`x`' is deduced remotely. That means that the most general type for function '`foo x`' is '`(Num a) => a -> a`'. The concrete type instance of '`a`' in the call '`foo 42`' or '`foo 42.0`' is the type of the particular function argument which needs to be an instance of '`Num`'. In the case above the types are '`Integer`' and '`Double`'. Haskell is based on the Hindley-Milner algorithm [5, 11] which accomplishes type inference. Consider the following example written in Haskell:

```
foo x y = (length x) + y
```

The polymorphic function '`length`' of the standard library returns the length of a list. The type of the function is `[a] -> Int`. The inference algorithm is based on two phases and the first phase assigns every variable and expression a new type variable or uses a known type.

$$\begin{aligned} \text{foo} &: \tau_1 \\ \text{x} &: \alpha_1 \\ \text{y} &: \alpha_2 \\ (\text{length x}) + \text{y} &: \tau_2 \\ \text{length x} &: \alpha \text{ list} \rightarrow \text{Int} \end{aligned}$$

Afterwards a set of constraints is created. The constraints are typed by the following relation: $\Gamma \vdash e : t \mid C$. This may be read as "expression e has type t if assumptions Γ hold under the constraints C". For the example above we only need two rules: function call and binary operator:

$$\textit{Function} \quad : \quad \frac{\Gamma \oplus \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\} \vdash e : t}{\Gamma \vdash (x_1, ..., x_n) \rightarrow e : (t_1, ..., t_n) \rightarrow t}$$

$$\textit{Operator} \quad : \quad \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 + e_2 : t}$$

A list of all constraint typing rules of the Hindley-Milner algorithm can be found in [13, p. 322]. Afterwards the rules are applied to construct equations:

$$\begin{aligned} \tau_1 &= (\mathrm{x} \times \mathrm{y}) \to \tau_2 \\ \tau_2 &= \mathrm{Int} \\ \alpha_1 &= \alpha \text{ list} \\ \alpha_2 &= \mathrm{Int} \end{aligned}$$

In the second phase, the equations are unified to find the most general unifier (mgu). The mgu is found by substitution σ. If no most general unifier is found, then no solution exists. That means if the system of term equations holds

$$\sigma(s_i) = \sigma(t_i).\forall i$$

then σ is the most general solution. Therefore, after substitution is done the following types are inferred:

$$\begin{aligned} \tau_1 &\mapsto \mathrm{Int} \\ \alpha_1 &\mapsto \alpha \text{ list} \\ \alpha_2 &\mapsto \mathrm{Int} \end{aligned}$$

From this on the function type of example '`foo x y`' in Haskell notation is `[a] -> Int -> Int`. Summarized Haskell or most functional programming languages support full type inference but this is not the case in C++0x with *auto* and *decltype*.

In C++98 and 03 the *auto* keyword specifies an object with automatic storage duration. That means that a variable is stored on the stack and as soon as the stack gets popped the variable gets destroyed. Since this is done by design the *auto* keyword does not have any effect on local variables. Therefore, the meaning of *auto* was dropped in favour of type inference. Code as e.g. '`auto int x;`' will not work with C++0x anymore [14].

3 The *decltype* keyword in-depth

In some situations an object should have the same type as another object. Via *decltype* it is possible to query the type of an object and use it somewhere else, for example:

```
int a = 42;
double b;
decltype(a) c;       // c will be of type 'int'
decltype(a + b) d;   // d will be of type 'double' since the
```

```
                    // outermost expression results to
                    // type 'double'
decltype(foo()) e;  // e will be of the same type as the
                    // return type of the function foo
```

This is also useful in cases where no initialisation is done and the *auto* keyword cannot be used.

```
decltype(a * b) c;
...
foo(&c);
```

Variable '`c`' is initialized in the body of function '`foo`'. Therefore, *auto* cannot be used in this example and the *decltype* keyword shows its strength. Otherwise if an object is initialized it does not make any difference if *auto* or *decltype* is used:

```
int a = 42;
auto b = a;          // b will be of type 'int'
decltype(a) c = a;   // c will be of type 'int'
```

But it is suggested to use *auto* instead of *decltype* in cases where a variable is initialized because it is not that verbose.

Trailing-return-types

One of the major benefits of *decltype* is in conjunction with return types. Consider a function which multiplies its operands:

```
template<class T>
T mul(T a, T b) {
        return a * b;
}
```

If the multiplication operator is overloaded with a different return type than the object itself, then the template will not work anymore since it expects the same type for the return value as for the function arguments. With *decltype* it is possible to make it generic again (s. appendix C for a complete example):

```
template<class T>
[] mul(T a, T b) -> decltype(a * b) {
        return a * b;
}
```

The return type of template function '`mul`' is now deduced from the expression '`a * b`'. This technique is known as *trailing-return-types*.

The proposal [9] also mentions so-called wrapper or forwarding functions as one of the main motivations for *decltype*. A forwarding or wrapper function calls another function e.g. with a different set of arguments but the return type has to be the same.

```
double foo (int    x) { return double(x); }
int    foo (double x) { return int(x);    }

template <class T> T bar (T x) {
        return foo (x);
}

int main (void) {
        foo (42);     // works
        foo (42.42);  // works
        bar (42);     // error
        bar (42.42);  // error
}
```

Function '`bar`' is a forwarding function for '`foo`'. Since '`foo`' has another return type than the type of its argument, template instantiation of '`bar`' will not work. With a *trailing-return-type* it is possible:

```
template <class T> [] bar (T x) -> decltype(foo(x)) {
        return foo (x);
}
```

In this case the return type is deduced from the return value of function '`foo(x)`' which depends on the argument '`x`'. With C++98 or 03 it is not possible to write forwarding functions in a general way which cover all possibilities without introducing extra template arguments.

During the last proposals the syntax of *trailing-return-types* changed. In proposal [8] the syntax was '`auto foo() -> decltype(bar)`' and changed to '`[] foo() -> decltype(bar)`' in [2] which conforms to lambda functions [7] of C++0x.

4 Compiler support

Compiler support for the keywords *auto* and *decltype* and also for the *trailing-return-types* is still in progress. Table 1 shows a summary of four common compilers. According to a blog entry from a Microsoft Developer [10] the upcoming release of Visual C++ will include support for *auto*, *decltype* and *trailing-return-types*. One reason why some vendors do not include support for C++0x features might be the fact that the standard is still a working draft.

Compiler suite	auto	decltype	trailing-return-type via '`auto`'	trailing-return-type via '`[]`'
GCC 4.4.1	yes	yes	yes	no
LLVM and Clang SVN checkout 2009-10-25	no	yes	no	no
Intel C++ Compiler Professional Edition 11.1	yes	yes	no	no
Microsoft Visual C++ 2008 Express Edition SP1	no	no	no	no

Table 1: C++0x type inference compiler support

5 Conclusion

The major benefit of type inference in C++0x is that *auto* may reduce verbosity and *decltype* introduces a possibility to write generic forwarding functions. Although, type inference in C++0x is not as powerful as in some other programming languages, it is still a useful and handy feature. Especially code verbosity is often mentioned when templates are used. This may change if *auto* is accepted and widely used. Also the possibility to provide generic function return types makes it easier to write template functions.

Some compiler vendors released a non-standard extension called *typeof* [4, § Referring to a Type with typeof] whose functionality is related to *decltype*. But since it is non-standard the feature may vary between compilers. The *auto* keyword may also be simulated by the boost library and the '`BOOST_AUTO`' macro [16]. But these are all extensions and are not standardized by the C++ committee.

All in all type inference of C++0x may help to write more straightforward code and since it is a compile time feature, the run-time system is not affected. But compared to the type inference system as e.g. in Haskell it is not that powerful. This maybe because it is based upon the template argument deduction which is more or less a set of rules while type inference of Haskell is based upon a more powerful algorithm. If it is possible to introduce a type inference feature in C++ as e.g. in Haskell is questionable since C++ is weakly typed.

A *auto* example 1

```
struct foo {
        typedef int return_type;
        static int f() { return 42; }
};

struct bar {
        typedef double return_type;
        static double f() { return 42.0; }
};

template<class T>
void func() {
        typename T::return_type x = T::f();
        auto y = T::f();
}

int main(void) {
        func<foo>();
        func<bar>();
}
```

B *auto* example 2

```
// function foo takes a two-dimensional array and
// returns a two-dimensional array
int (*(foo)(int a[2][2]))[2] {
        return a;
}

int main(void) {
        int a[2][2] = { {1,2}, {3,4} };

        int (*(*x)(int[2][2]))[2] = foo;
        auto y = foo;

        int (*b)[2] = x(a);
        int (*c)[2] = y(a);
}
```

C *decltype* example

```
template<class T>
T mul(T a, T b) {
        return a * b;
}

template<class T>
[] new_mul(T a, T b) -> decltype(a * b) {
        return a * b;
}

struct my_class {
        int operator* (my_class x) {
                return 42;
        }
};

int main (void) {
        my_class a, b;
        mul (42, 42);    // OK
        mul (a, b);      // ERROR, the return type is
                         // different than the type of
                         // the object itself
        new_mul (a, b);  // OK
}
```

References

[1] Walter E. Brown. A Proposal to Improve const_iterator Use from C++0X Containers. Technical Report N1674=04-0114, C++ Standards Committee, August 2004.

[2] Lawrence Crowl and Alisdair Meredith. Unified Function Syntax. Technical Report N2931=09-0121, C++ Standards Committee, July 2009.

[3] International Organization for Standardization. Working Draft, Standard for Programming Language C++. Technical Report N2960=09-0150, C++ Standards Committee, September 2009.

[4] Free Software Foundation, editor. *Using the GNU Compiler Collection: For gcc version 4.4.1.* July 2009.

[5] Roger Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[6] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2003.

[7] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 3). Technical Report N2529=08-0039, C++ Standards Committee, February 2008.

[8] Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype and Auto. Technical Report N1705=04-0145, C++ Standards Committee, September 2004.

[9] Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype. Technical Report N1978=06-0048, C++ Standards Committee, April 2006.

[10] Stephan T. Lavavej. decltype: C++0x Features in VC10, Part 3. `http://blogs.msdn.com/vcblog/archive/2009/04/22/decltype-c-0x-features-in-vc10-part-3.aspx`, April 2009.

[11] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[12] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[13] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] Daveed Vandevoorde and Jens Maurer. Removal of auto as a storage-class specifier. Technical Report N2546, C++ Standards Committee, February 2008.

[15] David Vandevoorde. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

[16] Arkadiy Vertleyb and Peder Holt. typeof reference. `http://www.boost.org/doc/libs/1_40_0/doc/html/typeof/refe.html`, 2005.