Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 383–390.

Syntax Check of Embedded SQL in C++ with Proto

Zalán Szűgyi, Zoltán Porkoláb

[a]Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
e-mail:lupin@ludens.elte.hu

Abstract

The SQL is the most frequently used language to manipulate databases. However SQL is not a general purpose programming language, thus it is not appropriate to develop applications with. In the most cases SQL statements are embed into another programming language, like C++, Java or C#, allowing us to describe the general algorithms with a general purpose language and the database specific ones with SQL. Beside of the obvious benefit this solution has some disadvantages too. The compiler of the general purpose language do not know the syntax of SQL, thus it is not able to perform syntactical and semantical checking on SQL code in compilation time. That way, the errors of SQL codes are figured out only in run time which lowers the quality of our code and makes debugging more difficult.

However the template facility of C++ programming language – using it in a special way – allows us to run algorithms in compilation time. This paradigm is called Template Metaprogramming. With proper metaprograms the C++ compiler can be forced to check the syntax of SQL statements.

In this paper we present a solution how perform SQL syntax checking of embedded SQL statements during the compilation of C++ source codes.

1. Introduction

There are several fields of software development where the main parts of projects are written in general-purpose programming languages, but some parts exist to handle in special way. For these purposes, the most usual and efficient strategy is applying domain specific languages (DSL) [8]. DSL is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique.

The general-purpose progamming languages allow to apply DSL code into its source code different way. The most common is to insert DSL source code into the

original source as a string literal, and a run-time interpreter will parse and execute it when the program is running. The C++ programming language behaves similarly. With this solution it is very easy to insert DSL code into the source code, but it is decreases the quality of the code: While the general-purpose compiler deals the DSL code as a string literal, during the compile time the possible syntactical errors will be hidden. They can be figured out only in run-time, causing either excpetions or even worse undeterministic or unexpected program behaviour.

In C++ programming language [11], while the general way is to handle DSL code as a string literal, it is possible to force the compiler to perform syntax check of DSL source code in compile time. Doing this we need to define the DSL itself and the DSL code in a special way relying on the template system of the language and Templata Metaprogramming technigues [1]. That way the possible errors in DSL source code are emitted in compile time, the error messages can be more clear and detailed, making the bugfix easier and faster.

Since the style of metaprograms in C++ programming language is unusual and difficult, it requires high programming skills to write. Besides, it is sorely difficult to find errors in it. However tools and libraries are available to help design, develop and debug them. The Boost 3rd party C++ library [7] provides a utilities for metaprogramming called Boost::MPL [12]. Also in Boost there are libraries to provide facilities to define DSLs which the C++ compiler can parse. These libraries are Boost::Spirit [13] and Boost::Proto [14]. Porkoláb et al. provided a metaprogram debugger tool [9] in order to help finding bugs.

The Structured Query Language (SQL) [6] is one of the most well known and widely used DSL languages for managing data in relational database management systems [2], and originally based upon relational algebra. Its scope includes data query and update, schema creation and modification, and data access control. There are several database systems and each of them supports the most known general purpsose programming languages, providing third party libraries, to access databases from source code. However these solutions also perform syntactical and semantical checks only in run-time.

In this paper we provide a solution how to force C++ compiler to perform syntax check on embedded SQL language. We implemented a subset of the SQL language called miniSQL based on Boost::Proto. In our solution the elements of the miniSQL grammar are valid C++ entities, thus they has a stronger connection with the language then just be a string literal.

The paper is organized as follows. In section 2 we give a short description of the template metaprogramming paradigm. The section 3 shows the definition of our language. We provide the implementation details in section 4. An example comes in section 6, and the section 7 presents the conclusion and future work.

2. Template Metaprograms

Those programs, which run at compile time are called template metaprograms [1]. Template metaprograms stand for the collection of templates, their instantiations

and specializations, and perform operations at compile time. The basic control structures like condition and iteration appear in them in a functional way [10]. The iterations in metaprograms are applied by recursion. The template metaprogramming is proved to be Turing complete [4]. See the following example where a template metaprogram computes the factorial of 5.

```
template<int N>
struct factorial {
  enum { value = N * factorial<N-1>::value };
};

template<>
struct factorial<0> {
  enum { value = 1 };
};

int main() {
  int result = factorial<5>::value;
}
```

To initialize the variable `result` here, the expression `factorial<5>::value` has to be evaluated. As the template argument is not zero, the compiler instantiates the general version of the `factorial` template with 5. The definition of `value` is `N * factorial<N-1>::value`, hence the compiler has to instantiate the `factorial` again with 4. This chain continues until the concrete value becomes 0. Then, the compiler choses the special version of `factorial` where the `value` is 1. Thus, the instantiation chain is stopped and the factorial of 5 is calculated. This algorithm runs while the compiler compiles the code. Hence, this example code is equivalent to `int result = 120`.

The condition is implemented by a template structure and its specialization.

```
template<bool cond_, typename then_, typename else_>
struct if_ {
  typedef then_ type;
};

template<typename then_, typename else_>
struct if_<false, then_, else_> {
  typedef else_ type;
};
```

The `if_` structure has three template arguments: a boolean and two abstract types. If the `cond_` is false, then the partly-specialized version of `if_` will be instantiated, thus the `type` will be bound by the `else_`. Otherwise the general version of `if_` will be instantiated and `type` will be bound by `if_`.

Since the style of metaprograms is unusual and difficult, it requires high programming skills to write. Besides, it is sorely difficult to find errors in it. Porkoláb et al. provided a metaprogram debugger tool [9] in order to help finding bugs.

3. Definition of miniSQL Language

In this chapter we present the EBNF [3] definition of our miniSQL language.

```
S ::= select_part from_part | select_part from_part where_part
select_part ::= SELECT g_S | select_part, g_S
from_part ::= FROM g_S | from_part, g_S
where_part ::=  WHERE condition_list
condition_list ::= condition | logical_and | logical_or
logical_and ::= condition_list AND condition_list
logical_or ::= condition_list OR condition_list
condition ::= g_S op g_S
op ::= == | < | <= | > | >= | !=
g_S ::= [A-z][A-z0-9]*
```

4. Implementation Details

Our language definition is based on Boost::Proto [14], which is a third party library to help defining DSLs as part of the C++ language. The DSLs defined that way are able to be checked by the C++ compiler, thus the possible error messages are detected in compile time. As the Boost::Proto is the base of our implementation, the terminal and non terminal symbols of the miniSQL grammar are special but valid types of C++, while the rules are defined by specialized template types. This approach implies that, all parts of the miniSQL grammar are C++ expressions.

This solution has an unusual consequence: the DSL must rely only on the original operators of C++. We can redefine their behaviour or exclude some of them, but it is not possible to introduce new ones or modify their precedences. Hereby the syntax of miniSQL slightly different as the original SQL syntax.

The following code snippet shows some rules of miniSQL grammar:

```
struct miniSQL_grammar :
    proto::or_<
        proto::plus<g_selectpart, g_frompart>,
        proto::plus<
            proto::plus<
                g_selectpart,
                g_frompart
            >,
            g_wherepart
       >
```

```
    > {};

struct g_condition_list :
    proto::or_<
        g_S,
        g_logical_and,
        g_logical_or
    > {};
```

The `struct miniSQL_grammar` is the start symbol. The template type `proto::or_` indicates a selection, thus the miniSQL grammar element either can be a `Select` and `From` concatenated by `operator+`, or can be `Select`, `From` and `Where`. The `proto::plus` template type refer to the `operator+`. The definition of other rules are similar, where `proto::shift_left` refers to the `operator«` and `proto::comma` to the `operator,`. The `g_S` nonterminal refers to the identifiers and the literals of the miniSQL. We present the way to handle these entities in the next chapter.

5. Handling Keywords, Identifiers and Literals

In our implementation of miniSQL all the grammar elements are valid C++ types. The operators are redefined C++ operators. The problem is that an identifier can be almost any sequence of alphanumerical characters. Predefining all of these identifier is impossible. The proto library provides special type which can refer to any kind of string literal in the embedded language. We can use that type until all the subexpression of our embedded language has at least one proto type as an argument. Otherwise the original operator will be applied. See the following code snippet:

```
struct g_identifier :
    proto::terminal<proto::convertible_to<std::string> > {};

struct g_literal :
    proto::terminal<proto::convertible_to<std::string> > {};

//...
"field_name" == "concrete_value" // (*)
```

This code snipped shows that every king of C++ language element which are convertible to std::string can be stand on the place of `g_identifier` or `g_literal`. The problem occures when we want tho write a restriction of an SQL statement for a given column in a `WHERE` clause. Both of the name of field and the value of restriction can be a string literal. See the marked line (*). While none arguments of the equality operator are Proto type the original `operator==` is applied instead of the `operator==` redefined in our grammar.

To solve this problem we implemented a wrapper class, which can wraps any string literals and turn their type into a Proto type. See the main parts of the wrapper classes below:

```
template<bool Complete>
struct _S : DisabledOperators<_S<false> >
{
    _S(const char* s) : str(s);
    _S(const std::string& s) : str(s);
    template <typename T>
    _S(const T& d);
    operator std::string() const;
    std::string str;
};

template<>
struct _S<false> : DisabledOperators<_S<false> >
{
    /* same constructors as in previous struct */
    _S<true> operator==(const _S<false>& rarg) const;
    _S<true> operator<(const _S<false>& rarg) const;
    /*... other necessary operators */
};

struct S : _S<false>;
```

The template `struct _S` is wrapper class. It can wrap any values which are convertible to `std::string`. The template argument `Complete` indicates whether this wrapper class is parsed during the evaluation of our miniSQL code.

While in C++ it is possible to chain equality operators (e.g.: a == b == c), we want to disable it in our language, therefore we defined the corresponding operators only in `struct _S` specialized to false. Since these operators returns `_S<true>` the next step in evlauation chain will fail.

The struct `DisabledOperators` provides better understandable error messages if someone tries to use inappropriate operators.

At last the `struct S` derived from `struct _S<false>` just to make wrapping easier.

The keywords are simple Proto types:

```
struct Select {};
struct From {};
struct Where {};

proto::terminal< Select >::type const SELECT = {{}};
proto::terminal< From >::type const FROM = {{}};
```

```
proto::terminal< Where >::type const WHERE = {{}};

//...
```

6. Example

In this chapter we present a small SQL query, where we want the receive name of the students whose studying at ELTE and living in Eger. The first code snippet is the query written in normal SQL syntax, and the second one is the same written in out miniSQL language.

```
SELECT name
FROM students
WHERE university = 'ELTE' and city = 'Eger';
```

```
(SELECT << "name") +
(FROM << "students") +
(WHERE << ( (S("university") == S("ELTE))
          -AND- ( S("city") == S("Eger")  ))
);
```

7. Conclusion and Future Work

In this article we presented an embedded language of C++, called miniSQL and it is a subset of SQL. The main feature of this language is the grammatical rules, the terminals and the nonterminals are valid C++ types and expressions. That allows us to force the C++ compiler to perform syntactical check of miniSQL in compile time. Hereby the possible errors turns out in compile time making the correction easier and more efficient. Our solution is based on Boost::Proto and template metaprogramming paradigm. Gil et al provided another solution to handle embedded sql statements in C++ [5].

Our future work is to extend our language to get the same expression power as SQL has. We would like to improve our parser to be able to perform some semantical check too. We plan to improve our error messages to provide more sophisticated information about errors.

References

[1] Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley (2004)

[2] Andrews, T., Harris, C.: Combining language and database advances in an object-oriented development environment. SIGPLAN Not. 22, 12 (Dec. 1987), 430-440

[3] Backus, J. W. et al.: Revised Report on the Algorithmic Language Algol 60

[4] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Reading (2000)

[5] Gil, J. (Y.), Lenz, K.: Simple and safe SQL queries with c++ templates, in proc. of Simple and safe SQL queries with c++ templates (2007), The ACM Digital Library pp. 13–24, (2007)

[6] Groff, J. R., Weinberg, P. N.: SQL, the complete reference.

[7] Karlsson, B.: Beyond the C++ Standard Library, An Introduction to Boost. Addison-Wesley, Reading (2005)

[8] Parr, T.: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers). Pragmatic Bookshelf, 2009.

[9] Porkoláb, Z., Mihalicza, J., Sipos, Á.: Debugging C++ Template Metaprograms, in proc. of Generative Programming and Component Engineering (GPCE 2006), The ACM Digital Library pp. 255–264, (2006)

[10] Sipos, Á., Porkoláb, Z., Pataki, N., Zsók, V.: Meta<Fun> – Towards a Functional-Style Interface for C++ Template Metaprograms, in Proceedings of 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007), pp. 489–502

[11] Stroustrup, B.: The C++ Programming Language Special Edition. Addison- Wesley, Reading (2000)

[12] `http://www.boost.org/doc/libs/1_42_0/libs/mpl/doc/index.html`

[13] `http://boost-spirit.com/home/`

[14] `http://www.boost.org/doc/libs/1_37_0/doc/html/proto.html`