From Abstract Parsing to Abstract Translation

Semen Grigorev
St. Petersburg State University
198504, Universitetsky prospekt 28
Peterhof, St. Petersburg, Russia
Email: rsdpisuy@gmail.com

Iakov Kirilenko
St. Petersburg State University
198504, Universitetsky prospekt 28
Peterhof, St. Petersburg, Russia
Email: jake@math.spbu.ru

***Abstract*—String-embedded language transformation is one of the problems which can be faced during database and information system migration. The conventional solution which is provided by a number of tools is based on run-time translation. We present a static *abstract translation* approach which originates from the *abstract parsing* technique [9] initially developed for syntax analysis of string-embedded languages. We present abstract translation algorithm and some optimization techniques, and discuss the results of its evaluation on a real-world industrial application.**

I. Introduction

Complex information systems are often implemented using more than one programming language. Sometimes this variety takes form of one *host* and one or few *string-embedded* languages. Textual representation of clauses in a string-embedded language is built at run time by a host program and then analyzed, compiled or interpreted by a dedicated runtime component (database, web browser etc.) Most general-purpose programming languages may play role of the host; one of the most evident examples of string-embedded language is dynamic SQL which was specified in ISO SQL standard in 1992 [7] and since then is supported by the majority of DBMS.

String-embedded languages may help to compensate the lack of expressivity of general-purpose language in a domain-specific settings or to integrate heterogeneous components of large system; however this approach comes with some price. In particular even the syntax analysis of string-embedded part of a system is undecidable in general case since its source code is represented implicitly using string-manipulation primitives, procedures and libraries, and generated “on the fly”. In a naïve implementation syntax analysis of embedded clauses is completely outsourced to the runtime environment which postpones many errors from being discovered prior to execution and thus compromises the ideas of code safety and static control.

Abstract parsing is the approach which was developed to overcome the aforementioned deficiency. In abstract parsing the source code of a host application is statically analyzed to provide some constructive representation of the set of string-embedded language clauses which can possible be generated at run time [9], [4]. This representation is then analyzed by a certain parsing algorithm which is usually derived from some existing one for plain strings [5]. Abstract parsing technique is utilized in a number of tools [8], [10], [2], [3] for program analysis and understanding.

While abstract parsing can help in application analysis it cannot handle the case of application *transformation*. As a practical use case for string-embedded language transformation we can mention reengineering. During reengineering it is sometimes necessary to migrate from one database management system to another; this migration may require a transformation of string-embedded clauses.

One of the options is dynamic translation at run time [1]. However this solution not always desirable. First, it may degrade the performance of the system due to introduction of extra processing stage. Next, with dynamic translation the ultimate goals of the reengineering are not achieved since some part of the original system escaped transformation.

Another approach includes translation of stored SQL which is supported by a number of existing production tools for database application development [11], [13], [12]. However, these tools do not support dynamic SQL translation and thus provide only partial solution.

The contribution of this paper is an approach for *abstract translation*. Similar to abstract parsing first we perform static analysis to build an approximation for the set of all generated clauses. Then our algorithm performs analysis which, unlike abstract parsing, produces *parsing forest* — a family of syntax trees, each of which represents the result of translation of certain input sequence. New correct assignments for all relevant string values in the host program are calculated then. Our approach works only when the source and the target languages are syntactically close enough (e.g. when they are two dialects of the same language). We discuss some heuristic which helps to reduce the complexity of the algorithm in many practical cases and present the results of its application for the migration of a real-world industrial project from MS-SQL Server 2005 to Oracle 11gR2 platform.

II. Graph-based Input Representation

As we mentioned above, the first stage of abstract parsing is static approximation of relevant string values. Two main representations for approximated values were used so far. In [4], [2], [3] the sets of potential string values are described using regular expressions; in [9] approximations are represented in more implicit form as a solutions of a system of (recursive) dataflow equations.

We did not find a way to scale either of these representations to abstract translation case. Instead, we represent the input stream for abstract translation via flow graph with one source and one sink nodes and string-labeled edges. The

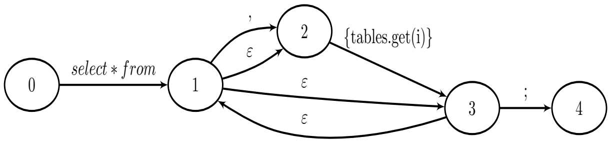

Fig. 1. Correct finite automaton (graph representation)

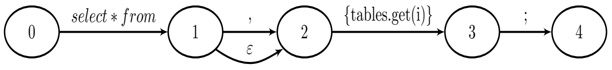

Fig. 2. Result of cycles approximation

labels of edges in this graph represent the results of constant propagation so that every path in input graph corresponds to one potential value of dynamic query. Moreover, we perform lexical analysis on graphs which converts the initial string-labeled graphs into graphs, labeled by tokens.

Since any cycle in the input graph generates infinite sequence of tokens which upon translation is turned into infinite forest we simplify the graph even more. We replace each cycle with the single repetition. For example in the order to get regular approximation of query value set from code presented below we should build the next regular expression:

$$\text{"select * from "} \cdot (\{\text{tables.get(i)}\}|\varepsilon)^* \cdot \text{";"}$$

and the corresponding finite automaton (see Fig. 1). Note that we do not care about approximation for *tables.get(i)* expression because it depends on a constant propagation algorithm. But we will get the graph where cycle replaced with only one repetition of its body. The result of such approximation is presented in Fig. 2.

```
query = "select * from ";
for(int i = 0; i < tables.size(); i++)
{
    if(i != 0) query += ", ";
    query += tables.get(i);
}
query += ";";
```

As you can see, in our example we do not produce strings like `select * from tbl1, tbl2;` or `select * from ;`. So, we can not check it. We process only two strings and all of them are in original infinite set: `select * from tbl1;` and `select * from, tbl1;`. As a result, we do not process all possible values but we process all variables used for query construction and it is enough for such tasks as code highlighting or transformations because all parts of expression are processed.

Thus the graph becomes cycle-free and we can process all vertices in the topological order. While this drastic simplification is completely heuristic our experience of dynamic SQL translation for real information systems showed that DAG is still a good approximation for practical use.

As an example consider the following code snippet:

```
(1) IF @X = @Y
(2) SET @TABLE = '#tbl1'
(3) ELSE
(4) SET @TABLE = 'tbl2'
(5) SET @S = 'SELECT x FROM ' + @TABLE
(6) EXECUTE (@S)
```

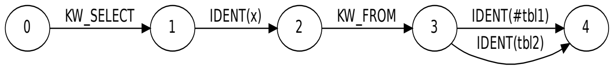

Fig. 3. Tokenized input graph

Variable `@S` contains dynamically generated query and can have two potential values at the point of query execution. During approximation we can build a graph which represents the set of potential values of the variable `@S` at the line 6. Each edge of this graph is labeled by a token which represents a part of the query (see Fig. 3).

Note that real-world systems can communicate with other systems source of which may be inaccessible to analyze. These systems can contain parts of queries to process and we should use some approximations. For example, clients applications of information system can sent conditions for filters (conditions for `where` clause of `select` statement) as part of requests.

III. Abstract Translation Algorithm

Our approach for abstract traslation borrows the idea of reusing the control structures used in classical parsing from [9]. Control tables of LALR analyzer may be generated by some conventional tool (e.g. yacc[1]). The interpreting automaton, however, has then to be modified to be able to compute all possible parser states for each vertex of the input graph.

For example, let we have the following grammar:

```
s -> Ae
e -> BD
e -> CD
```

An input graph is shown on the Fig. 4. The set of parser states for each vertex of the graph can be calculated during syntax analysis. The result of state calculation is shown on the Fig. 5.

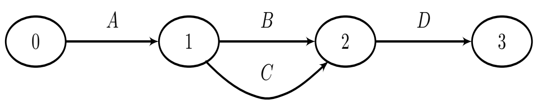

Fig. 4. Input graph for abstract parsing

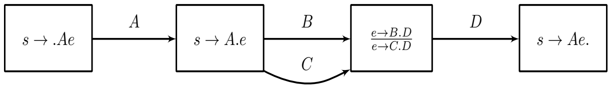

Fig. 5. Parser states

In the case of translation (not parsing) the parsing state consists of state of the automaton and some *semantic* value

[1] http://dinosaur.compilertools.net

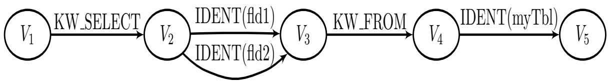

Fig. 6. Graph with states possible to merge

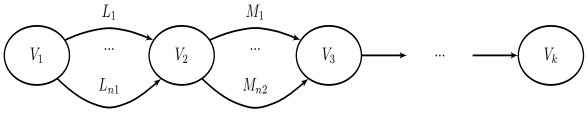

Fig. 7. Graph which requires an exponential resources for translation

which represents the result of translation built so far. In particular, the translation algorithm works not only with token types, but also token values.

One of the possible solution of translation is abstract parsing algorithm with mechanism of stack splitting for semantic calculation support. It disallows to merge states and creates a new copy of the whole stack for the each branch of the input graph.

However, this approach faces the exponential memory usage problem. For example parser states for vertex V_3 on the Fig. 6 should be equal for two input edges but if we want to calculate semantics, then we get two different states because identifiers has different values.

Queries which contain a huge number of branches is a big problem. The number of states is an exponential function of the number of branches because for each branch we should produce $n * k$ states where n is a number of states in the root of the fork vertex and k is the number of branches. One of the most frequent example of queries with big number of branches is `select` query. Each of fields to select can be calculated with if-statement or case-statement. Example of such graph is presented on the Fig. 7.

If we use only sequentially concatenated if-statements then the number of parsing trees is 2^n where n is a number of if-statements (or number of branches). In some real-world systems we have faced the queries which contains more than 100 branches. The full forest calculation by naive adaptation of abstract parsing is impossible for such queries.

We propose the following solution for the forest size minimization problem. We have previously mentioned that the result of translation is a new values for all variables which were used for queries construction. It is sufficient to construct not the full forest but only the minimal set of trees such that after translation every variable gets new value.This way, we can process not all paths in the input graph but only minimal set which contains all edges. Note that we cannot calculate this set prior to the parsing because we cannot be sure that every path produces syntactically correct value. If some path contains error than the tree for that path is not constructed and we may lose information about variables. For example consider the graph presented on the Fig. 8.

The one possible set of paths which we can calculate before syntax analysis is $\{(L_1; M_1); (L_2; M_2)\}$. But every path here contains syntax errors and the result forest would be empty instead of containing two trees. We should choose another set (for example $\{(L_1; M_2); (L_2; M_1)\}$) to get the correct result.

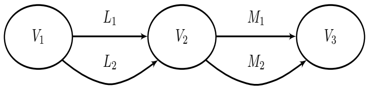

Fig. 8. Graph for minimal paths set selection.

So, path calculation is an iterative process. We perform state filtering during syntax analysis for each vertex with multiple input edges. Let describe the steps of the process:

- **Initial state.** Set of states for the vertex is empty.
- **Step.** For each step if the current vertex has multiple input edges then we should add new state to a state set for the current vertex if one of the following conditions is true:
  - new state corresponds to a path, which contains some edges which are not contained in any of the paths, which correspond to any state of the currently processing set;
  - new state corresponds to a parser state which is not yet presented in the currently processing set.

A pseudo code for the described algorithm is presented below.

```
/*
V  list of input graph vertices in the topological order.
v_s  start vertex of input graph.
*/

let filterStates v =
  let groupedByParserState =
    v.States.GroupBy (fun state -> state.Item)

  v.States = Set.empty

  for group in groupedByParserState do
    /* Each state corresponds with path from v_s to v.
     Set of paths specify set of edges of graph E_s.
     We should construct minimal set of paths which
     contains all edges of E_s. The next greedy algorithm
     can be applied to solve this problem.
     1) Order paths by length ascent.
     2) While current path contains edges which are not
     in the result set add this path in the result set.*/
     let ordered =
       group.OrderBy (fun s -> -1 * s.Path.Lenght)
     for s in ordered do
       if (s.Path contains edges which are
         not contained in any path corresponded
         with states from v.States || not s in v.States)
       then v.States.Add s

for v in V do
  v.States <-  /*step of syntax analysis*/
  /*If input degree of the vertex v more
    then 1 then try to filter states.*/
  if v.InEdges.Count > 1 then filterStates v
```

This way we can get state set which contains all parser states from input set but is not greater than it. Corresponding paths contain all possible edges in processed subgraph. Described algorithm of filtration allows to increase the performance of parsing by decreasing the number of parsing trees.

IV. Evaluation

We implemented our algorithm of abstract translation in a tool built on top of FsYacc[2]. We completely reused LALR generator, but implemented custom interpreter with stack copying ability.

Our tool was evaluated on a migration of a real-world project from MS-SQL Server 2005 to Oracle 11gR2. The original system contained 850 stored procedures and more than 3000 dynamic queries. The total size of the system was 2,7 million lines of code. More than half of all queries were complex; the number of query-generating operators varied from 7 to 212. The average number of query-generating operators was 40. We used PC workstation with Intel Core i7 2.6 GHz and 16 GB of RAM.

The results of comparison of two abstract translation implementations are presented in the Table I.

The first implementation was directly based on abstract parsing algorithm. That version was not adapted to process complex queries and turned system into active swapping. The analysis did not finish in acceptable time. Timeout (64 seconds) was added to limit one query processing time. Experiments showed that increasing timeout did not increase the number of processed queries. The number of queries, whose analysis was terminated by a timeout is shown in the table under the category "Dynamic SQL-queries with exponential growth of parsing forest".

The second implementation utilized state merging. State merging reduced the number of queries with exponential growth of parsing forest from 253 to 42, i.e. approximately in six times.

In the table below we present statistics for dynamic SQL query processing by two algorithms: original algorithm with timeout and algorithm with states merging.

Partially processed queries are those with non-empty parsing forest but with parsing or lexing errors. This category is the most difficult to deal with because error may be a false positive. Such situation may occur if query which triggers error can not actually be generated at run time.

V. Conclusion and Future Work

Semantics calculation for embedded languages is also the source of problems. The main problem is that we cannot guarantee semantics correctness during syntax analysis: we can get correct tree with incorrect semantic. Example of this situation is shown on Fig. 9. In presented graph we can choose 2 paths which contain all variables used for query value calculation. For example, let we choose the paths which produce the next queries: "`Select fld1 from myTbl1`" and "`Select fld2 from myTbl2`". Both chosen paths are syntactical correct but in the real system the table `myTbl1` may not contain the field `fld1`, and the table `myTbl2` may not contain the field `fld2`.

Also we have problems which correspond with syntax of analyzes language and its specification in documentation and grammar. For example, such clauses of `Select` statement as `group by` or `order by`. Any of these clauses can be omitted, but when the optional clauses are used, they must appear in the appropriate order and only one time per statement. But some simple approximation which allows to omit explicit enumeration of all variants of permutation is often used in the documentation and the grammar. Such approximation allows to accept input strings with arbitrary repetition of clauses (multiple repetition of one clause also possible). In the stored code such situation is not possible because this code should be correct but during graphs processing we can get `Select` query with multiple `group by` clause. This situation is not correct. The preferred solution of such problems is to use a special constructions in translation specification language. Also we can manually check correctness of parsing forest but this solution looks more difficult and less preferred.

TABLE I. Comparison of the original algorithm with timeout and the algorithm with state merging

Category description	Original algorithm with timeout	The algorithm with state merging
The total number of dynamic SQL queries	3122	3122
The number of successfully processed dynamic SQL queries	2181	2253
The number of partially processed dynamic SQL queries	408	522
Lexer errors	283	289
Parser errors	354	468
The number of not processed dynamic SQL queries	533	347
Lexer errors	140	134
Parser errors	280	305
Dynamic SQL queries with exponential growth of parsing forest.	253	42
Percentage of successfully processed dynamic SQL queries	69.86%	72.17%
Percentage of partially processed dynamic SQL queries	13.07%	16.72%
Percentage of dynamic SQL queries with non-empty forest	82.93%	88.89%

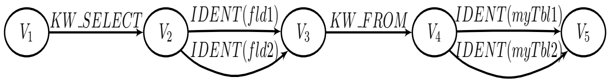

Fig. 9. All path in this graph are syntactical correct but semantics of some path may be incorrect.

References

[1] Shapot M., Popov E. Database reengineeing // Open Systems.DBMS. Number 4. 2004.

[2] Annamaa A., Breslav A., Kabanov J. e.a. An interactive tool for analyzing embedded SQL queries. Programming Languages and Systems. LNCS, vol. 6461. Springer: Berlin; Heidelberg. 2010. P. 131-138.

[3] Annamaa A., Breslav A., Vene V. Using abstract lexical analysis and parsing to detect errors in string-embedded DSL statements // Proceedings of the 22nd Nordic Workshop on Programming Theory. Marina Walden and Luigia Petre, editors. 2010. P. 20-22.

[4] Aske Simon Christensen, Mller A., Michael I. Schwartzbach. Precise analysis of string expressions // Proc. 10th International Static Analysis Symposium (SAS), Vol. 2694 of LNCS. Springer-Verlag: Berlin; Heidelberg, June, 2003. P. 1-18.

[5] Grune D., Ceriel J. H. Jacobs. Parsing techniques: a practical guide. Ellis Horwood, Upper Saddle River, NJ, USA, 1990. P. 322.

[2] http://fsharppowerpack.codeplex.com/wikipage?title=FsYacc

[6] Costantini G., Ferrara P., Cortesi F. Static analysis of string values // Proceedings of the 13th international conference on Formal methods and software engineering, ICFEM11. Springer-Verlag: Berlin; Heidelberg, 2011. P. 505-521.

[7] ISO. ISO/IEC 9075:1992: Title: Information technology Database languages SQL. 1992. P. 668.

[8] Java String Analyzer. URL: http://www.brics.dk/JSA/

[9] Kyung-Goo Doh, Hyunha Kim, David A. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using LR-parsing technology // Proceedings of the 16th International Symposium on Static Analysis, SAS09. Springer-Verlag: Berlin; Heidelberg, 2009. P. 256-272.

[10] PHP String Analyzer. URL: http://www.score.is.tsukuba.ac.jp/∼minamide/phpsa/

[11] PL/SQL Developer. URL: http://www.allroundautomations.com/plsqldev.html

[12] SQL Ways. URL: http://www.ispirer.com/products

[13] SwissSQL. URL: http://www.swissql.com/

[14] Xiang Fu, Xin Lu, Peltsverger B. e.a. A static analysis framework for detecting SQL injection vulnerabilities // Proceedings of the 31st Annual International Computer Software and Applications Conference. Vol. 01, COMPSAC07, Washington, DC, USA, IEEE Computer Society, 2007. P. 87-96.