Parsing of Regular Sets

Ekaterina Verbitskaia[1], Semen Grigorev[2], and Dmitry Avdyukhin[3]

[1] Saint Petersburg state university
kajigor@gmail.com,
[2] rsdpisuy@gmail.com
[3] dimonbv@gmail.com

Abstract. There is a class of applications which utilizes the idea of string embedding of one language into another. In this approach a host program generates string representation of clauses in some external language, which are then passed to a dedicated runtime component for analysis and execution. Despite providing better expressiveness and flexibility, this technique makes the behavior of the system less predictable since a whole class of verification procedures is postponed until run time, which complicates refactoring, testing and maintenance. We present a technique for syntax analysis, which works on regular approximations of a set of all dynamically-generated clauses and allows to ensure their well-formedness at compile-time. Our technique is based on a generalization of RNGLR algorithm, which, inherently, allows us to construct a finite representation of parse forest for regularly approximated set of input strings. This representation can be further utilized for semantic analysis and transformations in the context of reengineering, code maintenance, program understanding etc. The approach in question so far implements *relaxed parsing*: non-recognized strings in approximation set are ignored with no error detection.

Keywords: string-embedded language, string analysis, parsing, parser generator, RNGLR

1 Introduction

String expressions provide simple and flexible way to communicate heterogeneous components of a computer program. On the other hand, they cause various runtime errors (i.e. by incorrect syntax of constructed queries) and security issues (i.e. SQL injections). To address these issues, static analysis of string-embedded code could be used.

General approach of such analysis may be the following. First, one should analyse host program and find hotspots — code lines which use constructed string expression or send it to another component. Then the set of possible string expressions in the hotspot should be approximated: method proposed in [**?**] may be used. After that, operations similar to lexical analysis and then parsing should be performed to check syntactic correctness of possible expressions and to create structured and machine-readable representation of string-embedded code. Finally, when such representation is created, custom user semantics could be calculated and more complex analysis could be performed.

Host program is indeed a generator of strings, so we can say that it specify a language L. Each of generated string is supposed to be written in some reference language L_r. To check syntactic correctness of generated expressions means to ensure that $L \subset L_r$. Language inclusion problem is undecidable in general case, but is decidable, if L is regular and L_r is, for example, deterministic context-free language. As host program is usually written in a Turing-complete language, L could be recursively enumerable (type-0 in Chomsky hierarchy). In order to deal with decidable problem, it is rational to approximate L with a regular language L_a: paper [?] provide a method for it.

In this paper we propose a parsing algorithm which could be used in static analysis of string-embedded languages. The algorithm takes an automaton specifying language L_a and grammar specification of L_r as an input, and constructs a compact representation of parse forest for all syntactically correct strings of L_a or report an error, if every string is syntactically incorrect.

2 Related Work

Our parsing algorithm is based on a RNGLR-algorithm presented by Elizabeth Scott and Adrian Johnstone in [?]. In order to better understand the paper, a reader should be familiar to its principles of work, so we briefly describe RNGLR-algorithm in this section. Also we point out differences between our approach and existing tools which operate with regular approximation of string-embedded language since we use such type of approximation as input for our algorithm.

2.1 Regular Approximation of Sting-Embedded Language

Some tools are aimed to build high quality regular approximation. For example, Stranger [?] which use forward reachability analysis to compute over-approximation of all string values for program. Further analysis in Stranger is based on patterns detection in approximation or generation finite subset of strings for analyzing with standalone tools. Implementation of our algorithm may use such tools as input generators.

Paper [?] presents Java String Analyzer (JSA) — tool for static syntax correctness checking of embedded SQL statements. This tool build regular approximation with Mohri-Nederhof [?] algorithm and then check its inclusion into reference grammar without parsing and forest construction.

Our algorithm is inspired by Alvor [?] which apply GLR-based technique for syntax correctness checking of regular approximation. Key difference of our algorithm is building of parse forest finite representation.

2.2 RNGLR

RNGLR stands for Right-Nulled Generalized LR and is able to process all context-free grammars including ambiguous. Ambiguities of grammar produce

Shift/Reduce and Reduce/Reduce conflicts; the algorithm carry out all possible actions in such situations. The algorithm uses parser tables, each cell of which can contain multiple actions in case of conflicts.

RNGLR-algorithm uses Graph Structured Stack (GSS) — efficient representation of the set of stacks produced during conflict processing. GSS is an ordered graph, vertices of which corresponds to elements of classical stack and edges link sequential elements together. Each vertex can have multiple incoming edges and by means of it be shared between several stacks. Vertex is a pair (s, l), where s is a parser state and l is a level — position in an input string. Vertices in GSS are unique and there is no multiple edges. GSS construction routine is illustrated with pseudocode sample **??**: addVertex and addEdge functions.

The feature of RNGLR-algorithm which let it process all context-free grammars is a specific way of handling *right nullable* rules (i.e. rules of the form $\mathrm{A} \rightarrow \alpha\beta$, where β reduces to the empty string). That is, not only reductions for items $\mathrm{A} \rightarrow \alpha \cdot$ are applied, but also for the items of the form $\mathrm{A} \rightarrow \alpha \cdot \beta$, where $\beta \Rightarrow \epsilon$. Thus, reduction length — the number of symbols to be reduced to a nonterminal — may be less than or equal to the length of righthand side of the rule. There are also possible reductions of 0-length, also called as ϵ-reductions, corresponding to items of the form $\mathrm{A} \rightarrow \cdot$.

RNGLR-algorithm reads an input from left to right, one token at a time, and constructs levels of GSS sequentially for each position in the input. In the main loop of the algorithm for each token from the input, firstly, all possible reductions are applied (see reduce function in pseuducode sample **??**), and then the next token is shifted (see push function in pseuducode sample **??**).

3 Algorithm

Input of the algorithm is a reference grammar G with alphabeth of terminal symbols T and a finite automaton $(Q, \Sigma, \delta, q0, F)$, where $\Sigma \subseteq T$. RNGLR parser tables and some accessory information ($parserSource$ in pseudocode sample **??**) are generated by reference grammar G. Likewise RNGLR-algorithm, we associate GSS vertices with the position in the input, and in our case the position is a state of the input automaton. We construct the inner data structure by copying input automaton graph and extending vertex type with the following collections:

processed
GSS vertices, all the pushes for which are processed. This collection aggregates all GSS vertices associated with inner graph vertex.

unprocessed
GSS vertices, pushes for which are to be processed. This collection is analogous to $\mathcal{Q}$ from classic RNGLR-algorithm.

reductions
Queue which is analogous to $\mathcal{R}$ from classic RNGLR-algorithm: stores reductions to be processed.

passingReductionsToHandle
Pairs of GSS vertex and GSS edge to apply passing reductions along them.

Algorithm 1 RNGLR algorithm

```
1: function addVertex($level, state$)
2:     if GSS does not contain vertex $v = (level, state)$ then
3:         add new vertex $v = (level, state)$ to GSS
4:         calculate the set of shifts by $v$ and the next token and add them to $\mathcal{Q}$
5:         calculate the set of zero-reductions by $v$ and the next token and add them to $\mathcal{R}$
6:     end if
7:     return $v$
8: end function
9: function addEdge($v_h, level_t, state_t, isZeroReduction$)
10:    $v_t \leftarrow$ addVertex($level_t, state_t$)
11:    if GSS does not contain edge from $v_t$ to $v_h$ then
12:        add new edge from $v_t$ to $v_h$ to GSS
13:        if not $isZeroReduction$ then
14:            calculate the set of reductions by $v$ and the next token and add them to $\mathcal{R}$
15:        end if
16:    end if
17: end function
18: function reduce
19:    while $\mathcal{R}$ is not empty do
20:        $(v, N, l) \leftarrow \mathcal{R}.Dequeue()$
21:        find the set $\mathcal{X}$ of vertices reachable from $v$ along the path of length $(l-1)$, or length 0 if $l = 0$
22:        for all $v_h = (level_h, state_h)$ in $\mathcal{X}$ do
23:            $state_t \leftarrow$ calculate new state by $state_h$ and nonterminal $N$
24:            addEdge($v_h, v.level, state_{tail}, (l = 0)$)
25:        end for
26:    end while
27: end function
28: function push
29:    $\mathcal{Q}^{'} \leftarrow$ copy $\mathcal{Q}$
30:    while $\mathcal{Q}^{'}$ is not empty do
31:        $(v, state) \leftarrow \mathcal{Q}.Dequeue()$
32:        addEdge($v, v.level + 1, state, false$)
33:    end while
34: end function
```

Besides parser *state* and *level* (which is equal to the input automaton state), we store collection of passing reductions in GSS vertex. Passing reduction is a three-tuple $(startV, N, l)$, representing reductions which path passed through the GSS vertex. This three-tuple is very similar to the one describing reductions, but in this case l is a remaining length of the path. Passing reductions are stored in all vertices of the path except the first and the last during path searching in makeReductions function (see pseuducode sample **??**).

The general idea of the algorithm is to traverse input graph and sequentially construct GSS in the similar manner as RNGLR does. When deal with graph instead of linear stream, the next symbol means the set of terminals on outgoing edges of current vertex. This leads to slightly different process of push and reduce calculation: see line 9 in pseudocode sample **??** and lines 7 and 22 in pseudocode sample **??**. We use queue Q to control the order of input graph vertices processing. Every time new GSS vertex is added, zero reductions should be processed and then new tokens could be shifted, so corresponging graph vertex should be enqueueed for further processing. Adding of new GSS edge could produce reductions to handle, so input graph vertex with which tail of the added edge is associated should also be enqueueed. See details of GSS construction in pseudocode sample **??**. Reductions are applied along the paths in GSS, and if new edge which tail vertex have been in the graph before is added, then new paths will possibly be added which means some reductions would be lost. So it is necessary to recalculate those passing reductions: see applyPassingReductions function in pseudocode sample **??**.

Algorithm 2 Parsing algorithm

```
1: function PARSE($inputGraph, parserSource$)
2:     if $inputGraph$ contains no edges then
3:         if $parserSource$ accepts empty input then
4:             report success
5:         else
6:             report failure
7:         end if
8:     else
9:         ADDVERTEX($inputGraph.startVertex, startState$)
10:        $Q.Enqueue(inputGraph.startVertex)$
11:        while no $error$ have found and $Q$ is not empty do
12:            $v \leftarrow Q.Dequeue()$
13:            PROCESSVERTEX($v$)
14:        end while
15:        if $v_f$ is the vertex in the last level of GSS and its state is accepting then
16:            report success
17:        else
18:            report failure
19:        end if
20:    end if
21: end function
```

Algorithm 3 Single vertex processing

```
1: function processVertex($v$)
2:     makeReductions($v$)
3:     push($v$)
4:     applyPassingReductions($v$)
5: end function
6: function push($innerGraphV$)
7:     $\mathcal{U} \leftarrow$ copy $innerGraphV.unprocessed$
8:     clear $innerGraphV.unprocessed$
9:     for all $v_h$ in $\mathcal{U}$ do
10:        for all edge $e$ in outgoing edges of $innerGraphV$ do
11:            $push \leftarrow$ calculate next state by $v_h.state$ and the token on $e$
12:            addEdge($v_h, e.Target, push, false$)
13:            add $v_h$ in $innerGraphV.processed$
14:        end for
15:    end for
16: end function
17: function makeReductions($innerGraphV$)
18:    while $innerGraphV.reductions$ is not empty do
19:        $(startV, N, l) \leftarrow innerGraphV.reductions.Dequeue()$
20:        find the set of vertices $\mathcal{X}$ reachable from $startV$ along the path of length
   $(l-1)$, or 0 if $l = 0$; add $(startV, N, l-i)$ in $v.passingReductions$ where v is an
   i-th vertex of the path
21:        for all $v_h$ in $\mathcal{X}$ do
22:            $state_t \leftarrow$ calculate new state by $v_h.state$ and nonterminal $N$
23:            addEdge($v_h, startV, state_t, (l = 0)$)
24:        end for
25:    end while
26: end function
27: function applyPassingReductions($innerGraphV$)
28:    for all $(v, edge)$ in $innerGraphV.passingReductionsToHandle$ do
29:        for all $(startV, N, l) \leftarrow v.passingReductions.Dequeue()$ do
30:            find the set of vertices $\mathcal{X}$ reachable from $edge$ along the path of length
   $(l-1)$
31:            for all $v_h$ in $\mathcal{X}$ do
32:                $state_t \leftarrow$ calculate new state by $v_h.state$ and nonterminal $N$
33:                addEdge($v_h, startV, state_t, false$)
34:            end for
35:        end for
36:    end for
37: end function
```

4 Proof of Correctness

5 Construction of Parse Forest Finite Representation

The forest of parse trees can have infinite size in case of infinite number of paths in the input graph, so some finite representation could be helpful for

Algorithm 4 Construction of GSS

```
1: function addVertex($innerGraphV, state$)
2:     if $innerGraphV.processed$ or $innerGraphV.unprocessed$ contains vertex $v$ which state = $state$ then
3:         return $(v, false)$
4:     else
5:         $v \leftarrow$ create new vertex for $innerGraphV$ with state $state$
6:         add $v$ in $innerGraphV.unprocessed$
7:         for all $e$ in outgoing edges of $innerGraphV$ do
8:             calculate the set of zero-reductions by $v$ and the token on $e$ and add them in $innerGraphV.reductions$
9:         end for
10:        return $(v, true)$
11:    end if
12: end function
13: function addEdge($v_h, innerGraphV, state_t, isZeroReduction$)
14:     $(v_t, isNew) \leftarrow$ addVertex($innerGraphV, state_t$)
15:     if GSS does not contain edge from $v_t$ to $v_h$ then
16:         $edge \leftarrow$ create new edge from $v_t$ to $v_h$
17:         $Q.Enqueue(innerGraphV)$
18:         if not $isNew$ and $v_t.passingReductions.Count > 0$ then
19:             add $(v_t, edge)$ in $innerGraphV.passingReductionsToHandle$
20:         end if
21:         if not $isZeroReduction$ then
22:             for all $e$ in outgoing edges of $innerGraphV$ do
23:                 calculate the set of reductions by $v$ and the token on $e$ and add them in $innerGraphV.reductions$
24:             end for
25:         end if
26:     end if
27: end function
```

practical use. It is natural to use Shared Packed Parse Forest (SPPF) presented by Rekers [**?**] as such representation. SPPF is a directed graph which merge the nodes of derivation trees.

6 Future Work

References

1. Yu, Fang and Alkhalaf, Muath and Bultan, Tevfik and Ibarra, Oscar H.: Automata-based Symbolic String Analysis for Vulnerability Detection. Form. Methods Syst. Des. 44(1), 44–70 (2014)
2. Christensen, Aske Simon and Møller, Anders and Schwartzbach, Michael I.: Precise Analysis of String Expressions. Proceedings of the 10th International Conference on Static Analysis, 1–18 (2003)

3. Annamaa, A., Breslav, A., Kabanov, J., Vene, V.: An Interactive Tool for Analyzing Embedded SQL Queries. In: Ueda, K. (eds.) APLAS 2010. LNCS, vol. 6461, pp. 131–138. Springer, Heidelberg (2010)
4. Scott, E. and Johnstone, A.: Right Nulled GLR Parsers. ACM Trans. Program. Lang. Syst. 28(4), 577–618 (2006)
5. Rekers, J. G.: Parser generation for interactive environments. PhD Thesis. Amsterdam: Universty of Amsterdam. 174 p. (1992)
6. Asveld, Peter R. J., Nijholt, A.: The Inclusion Problem for Some Subclasses of Context-free Languages. Theor. Comput. Sci. 230(1&2), 247–256 (1999)
7. Grigorev, S., Verbitskaia, E., Ivanov, A., Polubelova, M., Mavchun, E.: String-embedded Language Support in Integrated Development Environment. Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia. 21:1–21:11 (2014)
8. Mohri M., Nederhof, M.-J.: Regular approximation of context-free grammars through transformation. In J.-C. Junqua and G. van Noord (eds.), Robustness in Language and Speech Technology, pp. 153–163. Kluwer Academic Publishers (2001)