Динамически формируемый код: синтаксический анализ контекстно-свободной аппроксимации

Дмитрий Ковалев, Семен Григорьев
Санкт-Петербургский Государственный Университет
dmitry.kovalev-m@ya.ru, semen.grigorev@jetbrains.com

Аннотация

Многие программы в процессе работы формируют из строк исходный код на некотором языке программирования и передают его для исполнения в соответствующее окружение (пример — dynamic SQL). Для статической проверки корректности динамически формируемого выражения используются различные методы, одним из которых является синтаксический анализ регулярной аппроксимации множества значений такого выражения. Аппроксимация может содержать строки, не принадлежащие исходному множеству значений, в том числе синтаксически некорректные. Анализатор в данном случае сообщит об ошибках, которые на самом деле отсутствуют в выражении, генерируемом программой. В данной статье будет описан алгоритм синтаксического анализа более точной, чем регулярная, контекстно-свободной аппроксимации динамически формируемого выражения, позволяющей снизить количество ложных ошибок.

Ключевые слова: синтаксический анализ, динамически формируемый код, контекстно-свободные грамматики, GLL, GFG, dynamic SQL, DSQL

Введение

Современные языки программирования общего назначения поддерживают возможность работы со строковыми литералами, позволяя формировать из них выражения при помощи строковых операций. Строковые выражения могут создаваться динамически, с использованием таких конструкций языка, как циклы и условные операторы. Данный подход широко используется, например, при формировании SQL-запросов к базам данных из программ, написанных на Java, C# и других высокоуровневых языках (листинг 1).

Недостаток такого метода генерации кода заключается в том, что формируемые выражения, с точки зрения компилятора, являются обычными

```
private void Example (bool cond) {
    string columnName = cond ? "name" : "address";
    string queryString =
        "SELECT id, " + columnName + " FROM users";
    Program.ExecuteImmediate(queryString);
}
```

Листинг 1: Динамически формируемый SQL-запрос

строками и не проходят статические проверки на корректность и безопасность, что приводит к ошибкам времени исполнения и усложняет разработку и сопровождение системы. Включение обработки динамически формируемых строковых выражений в фазу статического анализа осложняется тем, что такие выражения, в общем случае, невозможно представить в виде линейного потока, который принимают на вход традиционные алгоритмы лекического/синтаксического анализа.

Для решения данной проблемы были разработаны различные методы статического анализа множества значений формируемого выражения. Как правило, язык, на котором написана исходная программа, тьюринг-полон, что делает невозможным проведение точного анализа. Поэтому распространенным подходом является построение некоторой аппроксимации рассматриваемого множества. Ряд предложенных ранее решений использует для анализа *регулярную аппроксимацию* — множество строк, генерируемых программой, аппроксимируется сверху регулярным языком, и анализатор работает с его компактным представлением, таким как регулярное выражение или конечный автомат.

В магистерской диссертации [7] был описан алгоритм, позволяющий проводить синтаксический анализ регулярной аппроксимации (конечного автомата, представленного в виде графа) множества значений динамически формируемого выражения. Основой для данного алгоритма служит алгоритм обобщенного синтаксического анализа Generalized LL (GLL, [9]). Такой подход позволяет получать конечное представление леса разбора [8] корректных строк, содержащихся в аппроксимации множества значений выражения. Это представление может быть использовано для проведения более сложных видов статического анализа и для целей реинжиниринга.

В данной статье будет представлен алгоритм синтаксического анализа, который основан на описанной выше модификации GLL, но работает с более точной, чем регулярная, контекстно-свободной аппроксимацией множества значений динамически формируемого выражения. Использование более точной аппроксимации позволяет снизить количество ложных синтаксических ошибок, возникающих в результате того, что аппроксимирующее множество может содержать строки, осутствующие среди значений искомого выражения.

1 Обзор

Под *контекстно-свободной аппроксимацией* подразумевается грамматика, описывающая контекстно-свободный язык, который содержит в качестве подмножества возможные значения динамически формируемого выражения. Идея использования такой аппроксимации была заимствована из существующих работ в области анализа динамически формируемого кода, краткий обзор которых будет приведен в данной секции. Алгоритм синтаксического анализа регулярной аппроксимации на основе GLL был адаптирован нами для работы с графовым представлением КС-грамматик. В качестве такого представления мы использовали Grammar Flow Graph (GFG, [6]). Описания оригинального алгоритма и GFG также включены в обзор.

1.1 Подходы к анализу динамически формируемых выражений

Существует два основных подхода, позволяющих проводить различные виды анализа динамически формируемого кода. Один из них основан на проверке включения языка, аппроксимирующего множество значений динамически формируемого выражения, в эталонный контекстно-свободный язык, заданный пользователем. Данный подход был реализован в инструментах Java String Analyzer (JSA, [3]) и PHP String Analyzer (PHPSA, [5]). Получение аппроксимации в них реализовано следующим образом — из исходного кода программы извлекается набор dataflow-уравнений, описывающих значения строковых переменных, которые участвуют в генерации выражения. Эти уравнения затем интерпретируются как контекстно-свободная грамматика, описывающая аппроксимирующий язык. PHPSA работает непосредственно с такой грамматикой, используя эвристики (т.к. в общем случае задача о включении КС-языков неразрешима [2]); в JSA контекстно-свободный язык дополнительно аппроксимируется регулярным.

Другой подход заключается в проведении синтаксического анализа аппроксимации множества значений выражения. Такое решение позволяет не просто ответить на вопрос о включении языков, но и реализовать дополнительную функциональность, такую как вычисление семантики или рефакторинг. К методам, основанным на данном подходе, можно отнести алгоритмы синтаксического анализа регулярной аппроксимации на базе семейства GLR [1, 11] и GLL-алгоритмов [7], а также алгоритм абстрактного синтаксического анализа [4], позволяющий совместить синтаксический анализ с решением dataflow-уравнений, получаемых при помощи PHPSA.

1.2 Синтаксический анализ регулярной аппроксимации на основе GLL

Generalized LL (GLL) [9] — алгоритм синтаксического анализа, позволяющий, в отличие от классических LL-анализаторов, работать с произвольными контекстно-свободными грамматиками. При этом, GLL сохраняет такие

важные свойства алгоритмов нисходящего разбора, как интуитивная связь с грамматикой и простота отладки и диагностики ошибок.

Основной идеей GLL является использование дескрипторов, позволяющих полностью описывать состояние анализатора в текущий момент времени.

Определение 1 *Дескриптор — это четверка (L, u, i, N), где*

- *L — текущая позиция в грамматике вида* $A \to \alpha \cdot \beta$
- *u — текущая вершина Graph Structured Stack (GSS, [10])*
- *i — позиция во входном потоке*
- *N — построенный на данный момент узел дерева вывода*

В процессе работы поддерживается глобальная очередь дескрипторов. В начале каждого шага исполнения алгоритм берет следующий в очереди дескриптор и производит действия в зависимости от позиции в грамматике и текущего входного символа. Для обработки неоднозначностей в грамматике алгоритм добавляет дескрипторы для каждого возможного пути анализа в конец очереди. Результат работы алгоритма — множество деревьев разбора строки (лес разбора) — представляется в виде Shared Packed Parse Forest (SPPF) [8].

В рамках магистерской диссертации [7] на базе GLL был разработан алгоритм для синтаксического анализа регулярной аппроксимации множества значений динамически формируемого выражения. Под регулярной аппроксимацией здесь понимается детерминированный конечный автомат над алфавитом токенов (рис. 1c) . Оригинальный GLL-алгоритм был модифицирован для работы с нелинейным входом. Дескрипторы нового алгоритма хранят номер вершины входного графа вместо позиции в линейном потоке. Также, на шаге исполнения просматривается не единственный входной символ, а все ребра, исходящие из текущей вершины. Псевдокод данного алгоритма можно увидеть в приложении А. Основная логика работы представлена в функциях **dispatcher** (извлечение дескрипторов из очереди) и **processing** (анализ исходящих из текущей вершины ребер).

1.3 Grammar Flow Graph

Grammar Flow Graph (GFG, [6]) — связный помеченный граф, узлы которого соответствуют позициям в грамматике ($A \to \alpha \cdot \beta$). Различают следующие типы узлов (X — нетерминальный символ, t — терминальный):

- $A \to \alpha \cdot X\beta$ — call
- $A \to \alpha X \cdot \beta$ — return
- $A \to \cdot\alpha$ — entry
- $A \to \alpha\cdot$ — exit
- $A \to \alpha \cdot t\beta$ — scan

Для обозначения подграфа, представляющего продукции, имеющие в левой части нетерминал A, дополнительно используются узлы с метками $.A$ (start-узел) и $A.$ (end-узел). Подробное описание GFG можно найти в оригинальной статье.

Приведем определение выводимости строки в грамматике в терминах GFG. Для этого нам потребуется также определить понятие сбалансированного пути.

Определение 2 *Сбалансированным путем в GFG называется путь, подпоследовательность call и return-узлов которого сбалансирована.*

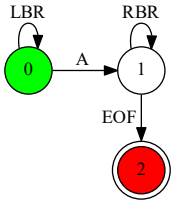
Определение 3 *Строка w выводима в грамматике, если в GFG существует сбалансированный путь из узла $.S$ в узел $S.$ (здесь S — стартовый нетерминал грамматики), и w может быть получена конкатенацией меток на ребрах, содержащихся в данном пути.*

```
x = "a"
while ...
    x = "[" x "]"
print x
```

(a) Код

$$
\begin{aligned}
s' &::= s \\
s &::= \texttt{LBR}\ s\ \texttt{RBR} \\
s &::= \texttt{A}
\end{aligned}
$$

(b) КС-аппроксимация



(c) Регулярная

Рис. 1: Исходный код и примеры аппроксимаций

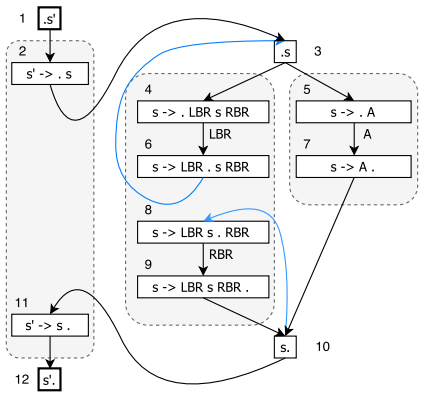


Рис. 2: GFG для грамматики 1b

2 Синтаксический анализ контекстно-свободной аппроксимации

Наш алгоритм принимает на вход управляющие таблицы LL-анализа, построенные по эталонной грамматике, и GFG, который является представлением КС-грамматики — аппроксимации множества значений выражения. Мы переиспользуем основные структуры данных и функции описанного ранее алгоритма анализа регулярной аппроксимации на основе GLL, расширяя данный подход для корректной обработки GFG.

Алгоритм последовательно обходит узлы GFG, производя синтаксический анализ порождаемых им строк. Для правильного построения таких строк, согласно определению выводимости строки в GFG-грамматике, для каждого просматриваемого пути необходимо поддерживать баланс call- и return-узлов. То есть, при прохождении пути алгоритм должен манипулировать дополнительным стеком (назовем его *CR-стеком*). При достижении call-узла в стек добавляется номер return-узла, соответствующего ему; при достижении end-узла необходимо снять со стека номер return-узла и продолжить обход из него.

Для экономии памяти мы не храним CR-стек для каждой из текущих ветвей работы алгоритма (напомним, что GLL-алгоритм может одновременно рассматривать несколько вариантов разбора строки). Вместо этого множество CR-стеков, по аналогии с основным стеков GLL-анализатора, представляется в виде GSS. Пример можно увидеть на рисунке 3. GSS позволяет хранить только одну копию общих префиксов нескольких стеков, каждый путь в нем соответствует отдельному CR-стеку.

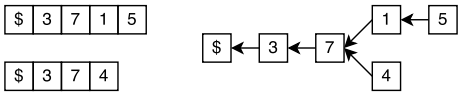


Рис. 3: Структурированный в виде графа стек

Для хранения указателя на текущую вершину стека мы добавили в дескрипторы дополнительное поле. Таким образом, дескриптором в нашем алгоритме называется пятерка вида (L, u, i, N, s), где i — номер вершины GFG, s — указатель на вершину CR-стека в GSS, остальные поля аналогичны тем, которые представлены в дескрипторах оригинального GLL-алгоритма.

Другой особенностью работы с GFG является то, что он, в отличие от регулярной аппроксимации — детерминированного конечного автомата, допускает возможность неоднозначного выбора пути обхода. Подобная ситуация возникает при наличии в исходной грамматике нескольких продукций, содержащих в левой части одинаковый нетерминал. Например, GFG на рисунке 2 содержит start-узел под номером 3, из которого выходит два ребра с пустой меткой (аналог ϵ-переходов в конечном автомате).

Механизм дескрипторов позволяет решать проблему недетерминированного выбора пути — для каждого из возможных вариантов создается от-

дельный дескриптор, который добавляется в очередь исполнения. Вернемся к примеру с узлом 3 на рисунке 2. Пусть в текущий момент времени мы имеем дескриптор $(L_1, u_1, i_1, N_1, s_1)$. При рассмотрении ребер, выходящих из узла 3, будут созданы дескрипторы $(L_1, u_1, 4, N_1, s_1)$ и $(L_1, u_1, 5, N_1, s_1)$. Если ранее такие дескрипторы не создавались (для контроля за этим в GLL поддерживается глобальное множество создаваемых дескрипторов), они будут добавлены в очередь.

Функции **dispatcher** и **add** (проверка и добавление в очередь дескриптора) алгоритма анализа регулярной аппроксимации были незначительно изменены нами для работы с расширенными дескрипторами. Функция **processing** и методы для работы с основным стеком и построения SPPF переиспользованы без изменений. Обработка start/call/exit-узлов и контроль за состоянием CR-стеков были реализованы во вспомогательной функции **closure**, псевдокод которой приведен ниже. Она исполняется перед вызовами **dispatcher** и **processing**, производя рекурсивный обход GFG до тех пор, пока не встретит start- или scan-узел. При достижении start-узла создаются дескрипторы для каждого из возможных путей, и управление переходит к **dispatcher**; scan-узел обрабатывается функцией **processing** так же, как и вершина конечного автомата в оригинальном алгоритме.

```
procedure CLOSURE
    tuple (C_L, C_u, C_i, C_N, C_S) is the current state of the analysis process
    v ← current GFG vertex
    if (v is not a final vertex) then
        switch VertexType(v) do
            case Start
                for (e ∈ v.OutputEdges) do
                    ADD(e.Target, C_L, C_u, C_N, C_S)
            case Call(r) where r is matching return vertex
                add CR-stack vertex w labeled r and edge from w to C_S
                C_S ← w
                C_I ← target vertex of v.OutputEdge
                CLOSURE
            case Exit
                let w be the target vertex of v.OutputEdge
                if (w is not a final vertex) then
                    C_I ← C_S.Label
                    C_S ← predecessor of C_S in CR-stack
                    CLOSURE
                else
                    C_I ← w
            case _
                return
```

Результатом работы алгоритма является SPPF, представляющий множество деревьев разбора всех строк, порождаемых GFG и одновременно с этим выводимых в эталонной грамматике.

3 Заключение

Описанный алгоритм был реализован на языке программирования F# в рамках проекта YaccConstructor. Исходный код доступен по ссылке: `https://github.com/YaccConstructor/YaccConstructor`. Результаты тестов показали, что использование контекстно-свободной аппроксимации позволяет увеличить точность синтаксического анализа динамически формируемого кода.

В дальнейшем планируется провести аппробацию алгоритма на реальных данных. Также, задача синтаксического анализа графов возникает в области биоинформатики, а именно ...

Список литературы

[1] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene. An interactive tool for analyzing embedded sql queries. pages 131–138. Programming Languages and Systems, Springer: Berlin, 2010.

[2] P. R. J. Asveld and A. Nijholt. The inclusion problem for some subclasses of context-free languages. *Theor. Comput. Sci.*, 230(1-2):247–256, 1999.

[3] A. S. Christensen, A. Møller, and M. I.Schwartzbach. Precise analysis of string expressions. pages 1–18. Proc. 10th International Static Analysis Symposium (SAS), Springer-Verlag: Berlin, June 2003.

[4] K.-G. Doh, H. Kim, and D. A. Schmidt. *Abstract LR-Parsing*, pages 90–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[5] Y. Minamide. Static approximation of dynamically generated web pages. pages 432–441. In Proceedings of the 14th International Conference on World Wide Web, WWW '05, ACM, 2005.

[6] K. Pingali and G. Bilardi. *A Graphical Model for Context-Free Grammar Parsing*, pages 3–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[7] A. Ragozina. Gll-based relaxed parsing of dynamically generated code. Master's thesis, SPbU, 2016.

[8] J. G. Rekers. *Parser generation for interactive environments*. PhD thesis, Citeseer, 1992.

[9] E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7), 2009.

[10] M. Tomita. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'85, pages 756–764, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[11] E. Verbitskaia, S. Grigorev, and D. Avdyukhin. *Relaxed Parsing of Regular Approximations of String-Embedded Languages*, pages 291–302. Springer International Publishing, Cham, 2016.

A Псевдокод модифицированного GLL

function DISPATCHER
  if $R.Count \neq 0$ **then**
    $(C_L, C_u, C_i, C_N) \leftarrow R.Dequeue()$
    $C_R \leftarrow \$$
    $dispatch \leftarrow false$
  else $stop \leftarrow true$

function PROCESSING
  $dispatch \leftarrow true$
  switch C_L **do**
    case $(X \rightarrow \alpha \cdot x\beta)$ where x is terminal
      for all $\{e | e \in input.OutEdges(C_i), e.Tag = x\}$ **do**
        $C'_N \leftarrow C_N, \quad C'_R \leftarrow C_R$
        if $C'_N = \$$ **then**
          $C'_N \leftarrow$ GETNODET(e)
        else
          $C'_R \leftarrow$ GETNODET(e)
        $C_L \leftarrow (X \rightarrow \alpha x \cdot \beta)$
        if $C'_R \neq \$$ **then**
          $C'_N \leftarrow$ GETNODEP(C_L, C'_N, C'_R)
        ADD$(C_L, C_u, e.Target, C'_N)$
    case $(X \rightarrow \alpha \cdot x\beta)$ where x is nonterminal
      $C_u \leftarrow$ CREATE$((X \rightarrow \alpha x \cdot \beta), C_u, C_i, C_N)$
      $slots \leftarrow \bigcup_{e \in C_i.OutEdges} pTable[x][e.Token]$
      for all $L \in slots$ **do**
        ADD$(L, C_u, C_i, \$)$
    case $(X \rightarrow \alpha \cdot)$
      POP(C_u, C_i, C_N)

function CONTROL
  while not $stop$ **do**
    if $dispatch$ **then** DISPATCHER
    else PROCESSING

function ADD(L, u, i, N)
  if $(L, u, i, N) \notin U$ **then**
    $U.Add(L, u, i, N)$
    $R.Add(L, u, i, N)$

function POP(u, i, z)
  if $u \neq u_0$ **then**
    $P.Add(u, z)$
    for all $(a, v) \in v.OutEdges$ **do**
      $y \leftarrow$ GETNODEP$(u.L, a, z)$
      ADD$(u.L, v, i, y)$

```
function CREATE(L, u, i, a)
    if (L, i) ∉ GSS.Nodes then
        GSS.Nodes.Add(L, i)
    v ← GSS.Nodes.Get(L, i)
    if (v, a, u) ∉ GSS.Edges then
        GSS.Edges.Add(v, a, u)
        for all (v, z) ∈ P do
            y ← GETNODEP(L, a, z)
            (_, _, k) ← z.Lbl
            ADD(L, u, k, y)
    return v
function GETNODET(e)
    tag ← e.Tag,  src ← e.Source,  tr ← e.Target
    if (tag, src, tr) ∉ SPPF.Nodes then
        SPPF.Nodes.Add(tag, src, tr)
    return SPPF.Nodes.Get(tag, src, tr)
function GETNODEP((X → ω1 · ω2), a, z)
    if ω1 is terminal or non-nullable nonterminal and ω2 ≠ ε then
        return z
    else
        if ω2 = ε then  t ← X
        else h ← (X → ω1 · ω2)
        (q, k, i) ← z.Lbl
        if a ≠ $ then
            (s, j, k) ← a.Lbl
            y ← findOrCreate SPPF.Nodes (n.Lbl = (t, i, j))
            if y does not have a child labeled (X → ω1 · ω2) then
                y′ ← newPackedNode(a, z)
                y.Chld.Add y′
                return y
            else
                y ← findOrCreate SPPF.Nodes (n.Lbl = (t, k, i))
                if y does not have a child labeled (X → ω1 · ω2) then
                    y′ ← newPackedNode(z)
                    y.Chld.Add y′
                    return y
    return SPPF.Nodes.Get(x, i, h)
```