Санкт-Петербургский Государственный Университет

Кафедра Системного Программирования

Байгельдин Александр Юрьевич

Оптимизация алгоритма лексического анализа динамически формируемого кода

Курсовая работа

Научный руководитель:
ст.пр., магистр ИТ Григорьев С. В.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Alexander Baygeldin

Optimization of lexical analysys of dynamically generated string expressions

Course Work

Scientific supervisor:
senior lecturer, master of IT Semyon Grigoriyev

Saint-Petersburg
2016

Оглавление

Введение

Динамически формируемый код — это код, который может быть получен и использован внутри другого кода при помощи строковых операций, таких как конкатенация, циклы, замена подстроки. Яркий пример такого кода — запросы SQL, которые составляются динамически в языках более общего назначения (C#, PHP и др.). На рис. 1 представлен пример кода, составленного с помощью условного выражения и конкатенаций.

```
1  private void Go(bool cond)
2  {
3    string tableName = cond ? "Sold" : "OnSale ";
4    string queryString =
5        "SELECT ProductID, UnitPrice, ProductName "
6            + "FROM dbo.products_" + tableName
7            + "WHERE UnitPrice > 1000 "
8            + "ORDER BY UnitPrice DESC;";
9    Program.ExecuteImmediate(queryString);
10 }
```

Рис. 1: Пример динамически формируемого кода

Однако в третьей строке примера при формировании имени таблицы был пропущен пробел, и эта ошибка будет выявлена только во время выполнения программы. Таким образом, при разработке и реинжиниринге систем, использующих динамически формируемый код, можно было бы избежать множество проблем, если бы в IDE существовала поддержка статического анализа подобного кода [2]. Лексический анализ является важным шагом такого статического анализа.

Задача лексического анализа — выделение лексем во входном потоке и сохранение привязки, т.е. позиции в тексте, к исходному коду. Часто при лексическом анализе используются инструменты для генерации лексических анализаторов по спецификации языка. Лексический анализатор переводит поток символов в поток лексем и может быть пред-

ставлен в виде конечного преобразователя. **Конечный преобразователь** (Finite State Transducer) — это математическая модель устройства, похожая на конечный автомат, с тем лишь дополнением, что каждому переходу сопоставляется дополнительное значение, которое выводится в выходной поток. На рис. 2 продемонстрирован пример лексического анализатора языка арифметических выражений (ради простоты, единственной операций является операция сложения), представленного в виде конечного преобразователя.

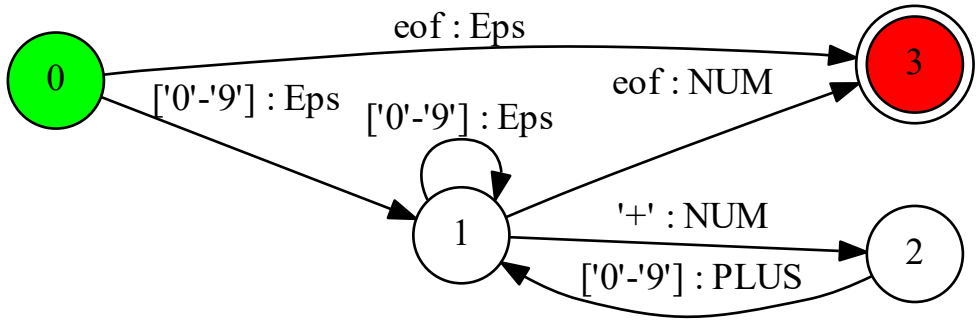

Рис. 2: Пример лексического анализатора, представленного в виде FST

Однако большинство инструментов для генерации лексических анализаторов могут работать лишь с линейным входом, что делает невозможным их непосредственное применение в нашем случае, т.к. поток символов формируется динамически. Одно из возможных решений [3] заключается в построении регулярной аппроксимации множества значений динамически формируемого выражения и последующем применении операции композиции [1] к двум конечным преобразователям: один их них построен по регулярной аппроксимации преобразованием автомата над строками в автомат над символами, а второй является классическим лексическим анализатором для языка, на котором написан динамически формируемый код. На рис. 4 изображен пример регулярной аппроксимации динамически формируемого кода, представленной в виде конечного автомата, построенного на основе кода на рис. 3.

```
1 private void Go(int cond){
2   string columnName = cond > 3 ? "X" : (cond < 0 ? "Y" : "Z");
3   string queryString = "SELECT name" + columnName + " FROM table";
4   Program.ExecuteImmediate(queryString);}
```

Рис. 3: Пример динамически формируемого кода

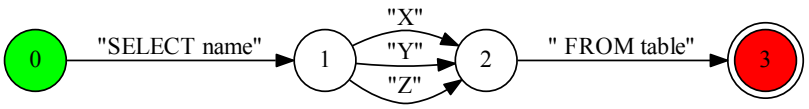

Рис. 4: Пример регулярной аппроксимации динамически формируемого кода

Результатом операции композиции конечных преобразователей является конечный преобразователь, который обладает тем свойством, что результат его работы совпадает с результатом последовательного применения участников композиции на том же входе. В нашем случае, первый конечный преобразователь переводит поток символов с их привязкой к исходному коду в поток символов, а второй (лексический анализатор) переводит поток символов в поток лексем. Таким образом, результирующий конечный преобразователь переводит поток символов с их привязкой к исходному коду в поток лексем, что и является целью лексического анализа. Результатом применения операции композиции к конечному преобразователю на рис. 5 и лексическому анализатору на рис. 2 является конечный преобразователь изображенный на рис. 6.

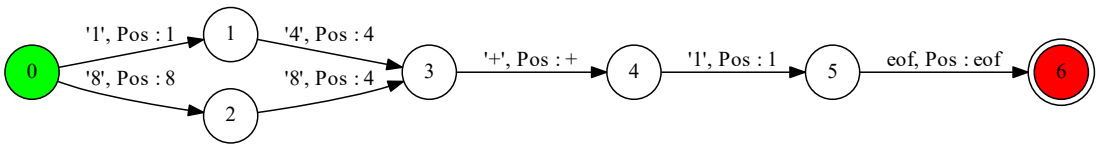

Рис. 5: Конечный преобразователь, участник операции композиции FST

В существующей реализации операция композиции является основной операцией в лексическом анализе динамически формируемых язы-

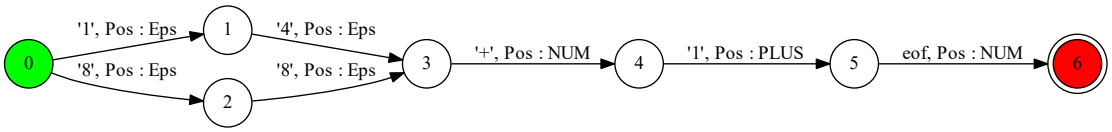

Рис. 6: Результат операции композиции FST

ков. Однако в проекте YaccConstructor [9, 8], в рамках которого было реализовано такое решение [3], ее производительность оказалась неудовлетворительной. Таким образом, целью данной работы являлось исследование возможности улучшения производительности лексического анализа за счет оптимизации алгоритма композиции конечных преобразователей.

1. Постановка задачи

Целью данной работы являлось исследование возможности улучшения производительности лексического анализа за счет оптимизации алгоритма композиции конечных преобразователей. Для достижения этой цели были поставлены следующие задачи.

- Исследовать алгоритмы композиции конечных преобразователей.
- Реализовать и интегрировать в проект YaccConstructor более оптимальный алгоритм композиции.
- Сравнить производительность реализаций текущего и выбранного алгоритмов.

2. Обзор

Конечный преобразователь может быть задан следующей шестеркой элементов: $\langle Q, \Sigma, \Delta, q_0, F, E \rangle$, где

- Q — множество состояний,
- Σ — входной алфавит,
- Δ — выходной алфавит,
- $q_0 \in Q$ — начальное состояние,
- $F \subseteq Q$ — набор конечных состояний,
- $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times Q$ — набор переходов.

Композицией двух конечных преобразователей $T_1 = \langle Q_1, \Sigma_1, \Delta_1, q_{0_1}, F_1, E_1 \rangle$ и $T_2 = \langle Q_2, \Sigma_2, \Delta_2, q_{0_2}, F_2, E_2 \rangle$ является конечный преобразователь $T = \langle Q_1 \times Q_2, \Sigma_1, \Delta_2, \langle q_{0_1}, q_{0_2} \rangle, F_1 \times F_2, E \cup E_\varepsilon \cup E_{i,\varepsilon} \cup E_{o,\varepsilon} \rangle$, где

- $E = \{\langle \langle p, q \rangle, a, b, \langle p', q' \rangle \rangle \mid \exists c \in \Delta_1 \cap \Sigma_2 : \langle p, a, c, p' \rangle \in E_1 \wedge \langle q, c, b, q' \rangle \in E_2\}$
- $E_\varepsilon = \{\langle \langle p, q \rangle, a, b, \langle p', q' \rangle \rangle \mid \langle p, a, \varepsilon, p' \rangle \in E_1 \wedge \langle q, \varepsilon, b, q' \rangle \in E_2\}$
- $E_{i,\varepsilon} = \{\langle \langle p, q \rangle, \varepsilon, a, \langle p, q' \rangle \rangle \mid \langle q, \varepsilon, a, q' \rangle \in E_2 \wedge p \in Q_1\}$
- $E_{o,\varepsilon} = \{\langle \langle p, q \rangle, a, \varepsilon, \langle p', q \rangle \rangle \mid \langle p, a, \varepsilon, p' \rangle \in E_1 \wedge q \in Q_2\}$.

Текущая реализация алгоритма композиции FST в проекте YaccConstructor работает по определению, т.е. перебирает всевозможные комбинации переходов FST участников композиции и добавляет в результирующий FST ребра, удовлетворяющие формальным описаниям из определения композиции. Поэтому временная сложность текущего алгоритма представляется как:

$$O(E_1 * E_2)$$

где E — число ребер соответствующего FST.

Для удобства эту временную сложность можно представить в следующем виде:

$$O(V_1 * V_2 * D_1 * D_2)$$

где V — число вершин, а D — максимальное число исходящих ребер соответствующего FST.

Важно заметить, что в результате работы алгоритма, несмотря на его корректность, в результирующем FST образуется множество недостижимых вершин, которые в дальнейшем с целью улучшения производительности, приходится удалять.

На замену текущему алгоритму был предложен алгоритм, представленный в статье [1], потому что в результате его работы не образуется недостижимых вершин и он обладает лучшей асимптотической сложностью:

$$O(V_1 * V_2 * D_1 * (log(D_2) + M_2))$$

где M — степень недетерминированности (максимальное количество исходящих из вершины ребер по одной метке).

Такая сложность достигается за счет поддержания очереди из пар вершин, в которых первая и вторая часть пары — вершины первого и второго FST соответственно. На каждой итерации алгоритма из очереди снимается одна пара вершин и перебираются комбинации ребер смежных этим вершинам. Далее, если метка выходного потока на ребре первого FST совпадает с меткой входного потока на ребре второго FST, то пара из вершин, в которые входят эти ребра, добавляется в очередь, а новое правило перехода добавляется в результирующий FST. Псевдокод алгоритма представлен в листинге (1).

Алгоритм возвращает новый FST T, множеством начальных вершин в котором является I, множеством конечных вершин — F, а множеством правил перехода — E. За основную очередь алгоритма обозначено K, а за Q — множество посещенных вершин, которое помогает

$Q \leftarrow I_1 \times I_2$
$K \leftarrow I_1 \times I_2$
while $K \neq \emptyset$ **do**
  $q = (q_1, q_2) \leftarrow Head(K)$
  $Dequeue(K)$
  if $q \in I_1 \times I_2$ **then**
    $I \leftarrow I \bigcup \{q\}$
  end
  if $q \in F_1 \times F_2$ **then**
    $F \leftarrow F \bigcup \{q\}$
  end
  for *each* $(e_1, e_2) \in E[q_1] \times E[q_2]$ *such that* $output[e_1] = input[e_2]$ **do**
    $q' = (target[e_1], target[e_2])$
    if $q' \notin Q$ **then**
      $Q \leftarrow Q \bigcup \{q'\}$
      $Enqueue(K, q')$
    end
    $E \leftarrow E \bigcup \{(q, input[e_1], output[e_2], q')\}$
  end
end
return T

Алгоритм 1: Композиция FST

предотвратить зацикливание алгоритма на входах с циклами.

3. Реализация

3.1. Реализация алгоритма

Алгоритм был реализован на языке F# семейства .NET в библиотеке для работы с конечными преобразователями YC.FST [3]. Можно выделить следующие особенности реализации.

- Класс конечных преобразователей в этой библиотеке поддерживает только целые числа в качестве состояний, но не кортежи из целых чисел, как это представляется в псевдокоде алгоритма. Поэтому было принято решение отображать сжатые представления кортежей из целых чисел в целые числа.

- Для проверки корректности работы алгоритма были написаны модульные тесты, в которых производится композиция и сравнение FST, составленных вручную.

3.2. Интеграция

В процессе работы была произведена реорганизация проекта YaccConstructor, одной из целей которой являлось получение возможности сравнения производительности текущего и выбранного алгоритмов. В рамках интеграции были выполнены следующие задачи.

- Библиотека YC.FST была интегрирована в библиотеку для работы с графами QuickGraph [7].

- Был произведен рефакторинг, заключавшийся в подмене сторонней библиотеки QuickGraph в YaccConstructor на использование собственной сборки этой библиотеки.

- Библиотека для работы с позициями в тексте YC.Utils.SourceText [4] была интегрирована в проект YaccConstructor с целью ее дальнейшего использования при оптимизации алгоритма лексического анализа.

4. Экспериментальное исследование

Сравнение производительности было произведено на заранее построенных регулярных аппроксимациях, построенных по реальному коду, в котором запросы T-SQL (Transact-SQL) формируются динамически, и лексическом анализаторе T-SQL. Результаты измерений в таблице 1 показывают, что реализация выбранного алгоритма дает существенный прирост в производительности как на небольших синтетических примерах, так и на реальном коде, регулярные аппроксимации которого содержат большое количество ребер и вершин.

Кол-во вершин в регулярной аппроксимации	Кол-во ребер в регулярной аппроксимации	Время работы текущего алгоритма (мс)	Время работы выбранного алгоритма (мс)
2	1	74	4
8	34	133	7
40	140	676	39
215	895	4045	248
310	687	7184	394
250	738	16526	1133
711	1766	30285	2068

Таблица 1: Сравнение производительности алгоритмов композиции FST, построенных по регулярным аппроксимациям динамически формируемых выражений на языке T-SQL

На выборке из 600 примеров, математическое ожидание ускорения (в количестве раз) выбранного алгоритма по сравнению с текущим равно 18.7, а среднее квадратичное отклонение равно 3.23. Однако стоит заметить, что в специфике нашей задачи выбранный алгоритм должен давать константный выигрыш в производительности для определенного языка по сравнению с текущим вне зависимости от входных данных. Но задачей данной работы являлось сравнение именно реализаций алгоритмов, а они отличаются техническими оптимизациями и необходимостью производить удаление недостижимых вершин после работы текущего алгоритма, в результате чего выигрыш в производительности перестает быть константным.

Заключение

В ходе выполнения данной работы были получены следующие результаты.

- Исследованы два различных алгоритма композиции FST.
- Алгоритм, обладающий лучшей производительностью, реализован и интегрирован в проект YaccConstructor.
- Произведено сравнение производительности реализаций алгоритмов композиции FST.
- Результаты работы представлены на конференции «Современные технологии в теории и практике программирования». Тезисы данной работы были опубликованы в сборнике материалов конференции.

Код, написанный в ходе выполнения данной работы, можно найти в репозитории проекта YaccConstructor [5]. В указанном репозитории автор принимал участие под учетной записью baygeldin.

В дальнейшем планируется исследование структур данных и алгоритмов для работы конечными преобразователями и позициями в тексте с целью улучшения производительности алгоритма лексического анализа. Также планируется полная поддержка и интеграция с инструментом для генерации лексических анализаторов FsLex [6].

Список литературы

[1] Mohri Mehryar. Handbook of Weighted Automata. — Monographs in Theoretical Computer Science. Springer, 2009. — P. 213–254. — URL: http://www.cs.nyu.edu/~mohri/pub/hwa.pdf.

[2] String-embedded language support in integrated development environment. / Semen Grigorev, Ekaterina Verbitskaia, Andrei Ivanov et al. // In Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '14). — 2014.

[3] Полубелова Марина. Лексический анализ динамически формируемых строковых выражений. // Дипломная работа кафедры системного программирования СПбГУ. — 2015. — URL: http://se.math.spbu.ru/SE/diploma/2015/bmo/444-Polubelova-report.pdf.

[4] Репозиторий проекта YC.Utils.SourceText. — URL: https://github.com/YaccConstructor/YC.Utils.SourceText.

[5] Репозиторий проекта YaccConstructor. — URL: https://github.com/YaccConstructor/YaccConstructor.

[6] Сайт проекта FsLex. — URL: http://fsprojects.github.io/FsLexYacc/.

[7] Сайт проекта QuickGraph. — URL: http://yaccconstructor.github.io/QuickGraph/.

[8] Сайт проекта YaccConstructor. — URL: http://yaccconstructor.github.io.

[9] Я.А. Кириленко, С.В. Григорьев, Д.А. Авдюхин. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем. // Дипломная работа кафедры системного программирования СПбГУ. — 2013. — URL: http://ntv.spbstu.ru/fulltext/T3.174.2013_11.PDF.