Разработка библиотеки комбинаторов запросов к графам

Шушаков Даниил Сергеевич

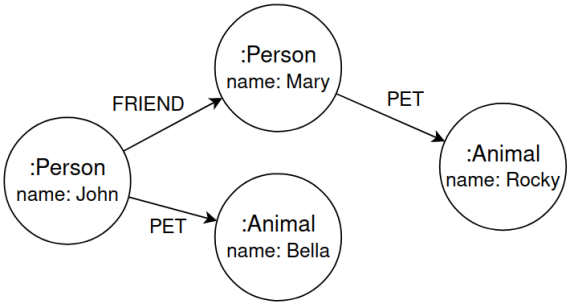
Научный руководитель: Григорьев Семен Вячеславович

Университет ИТМО

2 июня 2024 г.

Графовое представление данных

- Данные могут представляться в виде графа
- Хранятся в графовых базах данных
- Вершины и ребра могут иметь свойства вида «ключ: значение»



Языки запросов

- Для доступа к данным используются языки запросов, например, Cypher для Neo4j[1]
- В пользовательском коде они могут использоваться как строки:

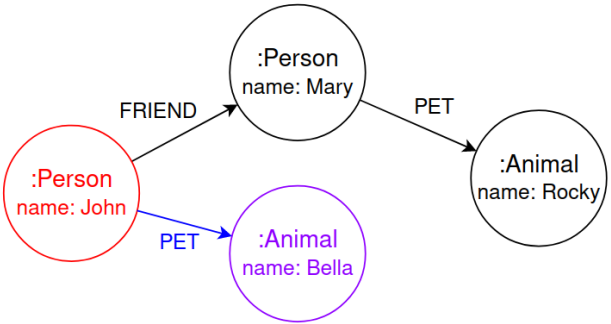
```
tx.execute("MATCH (n {name: 'my node'}) RETURN n, n.name")
```

- Недостатки:
  - Отсутствие проверки корректности запроса на этапе компиляции
  - Сложность составления и переиспользования подзапросов
- Удобно иметь бесшовную интеграцию запросов в язык программирования

[1] https://neo4j.com/docs/getting-started/get-started-with-neo4j/graph-database

Применение парсер-комбинаторов к графам

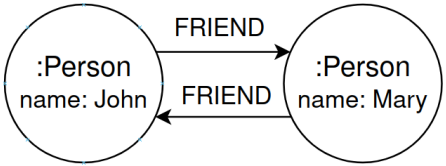
- Запрос к графу — нахождение путей, удовлетворяющих ограничениям
- Ограничения на пути можно описать с помощью грамматики
- Грамматику в ЯП общего назначения удобно описать через парсер-комбинаторы




```
val john = v { it.name == ”John” }
val pet = outE { it.label == ”PET” }
val s = john seq pet seq outV()
```

Дополнительные требования

- Описывать КС-ограничения: существуют применения в области биоинформатики[1], статическом анализе[2] и т.д.
- Разбирать циклы: результатов может быть бесконечно много



[1] Sevon et al (2016). Subgraph Queries by Context-free Grammars
[2] Reps (1997). Program analysis via graph reachability

Аналоги

Свойства	**Trails**[1]	**Meerkat**[2]
Сложность	Экспоненциальная	Полиномиальная
Левая рекурсия	Нет	Да
Циклы в графе	Ограничено	Да
Типизация	Контролирует корректность парсера	Возможны некорректные парсеры
ЯП	Scala	Scala

[1] Kröni et al (2013). Parsing graphs: applying parser combinators to graph traversals
[2] Izmaylova et al (2016). Practical, General Parser Combinators

Подходы к реализации

1. Meerkat основан на подходах из статьи Practical, General Parser Combinators[1]
   1. Мемоизация результатов парсера — левой рекурсия в грамматиках
   2. Генерация сжатого представления деревьев разбора (SPPF) — циклы в графах и полиномиальное время
2. Trails имеет удобные комбинаторы, контролирующие корректность парсеров

[1] Izmaylova et al (2016). Practical, General Parser Combinators

Цель и задачи

Цель: разработать библиотеку комбинаторов запросов к графам на основе алгоритмов, предложенных в работе Practical, General Parser Combinators, позволяющую описывать корректные парсеры.

Задачи:

1. Разработать базовую структуру парсера и комбинаторов с корректной типизацией
2. Поддержать левую рекурсию в грамматике
3. Поддержать разбор циклов в графе
4. Сравнить скорость работы с другими решениями

Реализация: базовая структура парсера

Решение разработано на языке Kotlin.

```
type Parser = (In) -> ParserResult<Out, Res>
```

- Парсер принимает входящее состояние, возвращает множество выходящий состояний и результатов
- Состояние: вершина, ребро, стартовое
- Базовые парсеры: `v, edge, inE, outE, inV, outV`
- Комбинаторы: `seq, or, many` и т.д.
- Комбинаторы запрещают объединять парсеры с конфликтующими типами состояний

Левая рекурсия: мемоизация результатов

- Разбор может бесконечно выполняться: левая рекурсия
- Хотим, чтобы вычисление парсера исполнялось один раз
- Парсер в качестве результата теперь возвращает вычисление:

```
type Parser = (In) -> (Continuation[1]<Out, Res>) -> Unit
```

- Парсер мемоизирует вычисления для каждого входящего состояния
- Вычисление запоминает все результаты и продолжения, при появлении нового результата вызывает эти продолжения
- Всем продолжениям будет переданы все результаты вычисления

[1] Можно воспринимать как callback

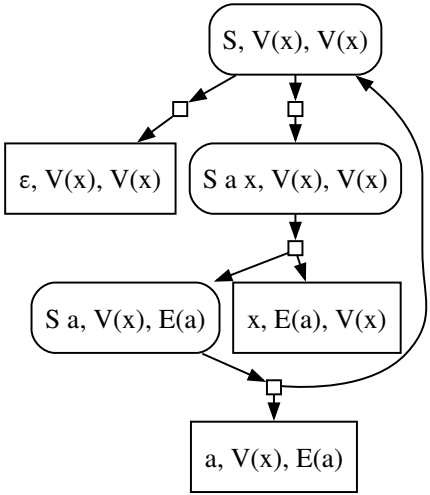
Циклы в графе: генерация SPPF

- Разбор графа с циклами потенциально может генерировать бесконечно много результатов
- Нужно представить результат такого разбора в виде конечной структуры — SPPF
- Результаты из SPPF извлекаются обходом

$$S \to \varepsilon \mid S\ a\ x$$

Грамматика

Граф-петля



Полученный SPPF

Набор данных для исследования времени работы

- Для исследования добавлена поддержка базы данных Neo4j
- Взяты три набора данных: онтологии[1], SFPQ_Data[2], Yago[3]
- Для первых двух использовался одинаковый КС-запрос

$$Q_1 \to subclassof^{-1}\ Q_1?\ subclassof$$
$$Q_1 \to type^{-1}\ Q_1?\ type$$

- Для Yago использовался регулярный запрос

```
MATCH
(x)-[:P1]->()-[:P2]->()-[:P3*1..]->()-[:P4*1..]->(:Entity{id:'42'})
RETURN DISTINCT x
```

[1] Zhang et al (2016). Context-free path queries on RDF graphs
[2] https://formallanguageconstrainedpathquerying.github.io/CFPQ_Data/graphs
[3] https://gitlab.inria.fr/tyrex-public/rlqdag

Результаты исследования времени работы

Граф		Кол-во вершин	Кол-во рёбер	Мое реш. (мс)	Meerkat (мс)
онтологии	atom-primitive	291	685	**19.5**	22.0
	b.-m.-p.[1]	341	711	**23.5**	42.5
	generations	129	351	**1.3**	1.4
	pizza	671	2,604	**61.4**	123.9
SFPQ_Data	enzyme	48,815	86,543	121	**116**
	eclass	239,111	360,248	1220	**1095**
	go	582,929	1,437,437	6231	**5367**
	go_hierarchy	45,007	490,109	12015	**8337**
yago		42,832,856	62,643,951	**5558**	OOM

[1] biomedical-mesure-primitive

Результаты

- Разработаны базовая структура парсера и комбинаторов, проверяющая корректность во время компиляции
- Поддержаны леворекурсивные грамматики
- Поддержан разбор циклов в графе
- Реализована базовая поддержка Neo4j графов
- Проведено исследование времени работы в сравнении с Meerkat:
  - На первом датасете мое решение быстрее в среднем на 28%
  - На втором датасете Meerkat быстрее в среднем на 15%
  - На третьем датасете только мое решение успешно выполнилось

Аналоги: Meerkat

- Meerkat — изначально библиотека с комбинаторами для разбора текста, реализующий алгоритм из статьи [1]
- В последствии расширена поддержкой запросов к графам[2]
- Т.к. изначально разрабатывалась для разбора текста, в ней парсеры принимали позицию в тексте
- В расширении позиция в тексте семантически поменялась на номер вершины в графе

[1] Izmaylova et al (2016). Practical, General Parser Combinators
[2] Verbitskaia et al (2018). Parser combinators for context-free path querying

Сигнатуры комбинаторов

```
fun BaseParser<I, O1, R1>.seq(p2: BaseParser<O1, O2, R2>)
    : BaseParser<I, O2, Pair<R1, R2>>

fun BaseParser<I, O, R>.or(p2: BaseParser<I, O, R>)
    : BaseParser<I, O, R>

fun rule(first: BaseParser<I, O, R>, vararg rest: BaseParser<I, O, R>)

fun BaseParser<I, O, A>.using(f: (A) -> B): BaseParser<I, O, B>

fun BaseParser<I, O, R>.that(constraint: BaseParser<O, O2, R2>)
    : BaseParser<I, O, R>

val BaseParser<S, S, R>.many: BaseParser<S, S, List<R>>
val BaseParser<S, S, R>.some: BaseParser<S, S, List<R>>
```

Комбинаторы that и using

Использование комбинаторов `that, using`:

```
val mary = outV { it.value == ”Mary” }
val loves = outE { it.label == ”loves” }
val friend = outE { it.label == ”friend” }
val maryLover = v().that(loves seq mary)
val p = maryLover seqr (friend seq outV()).some
val s = p using {edges -> Pair(edges.size, edges.last().second)}
```

Запросы для исследования времени (КС)

```
val subclassof1 = throughInE { it.label == "subClassOf" }
val subclassof = throughOutE { it.label == "subClassOf" }
val type1 = throughInE { it.label == "type" }
val type = throughOutE { it.label == "type" }
val Q1 = LazyParser<Neo4jVertexState, Neo4jVertexState, Any>()
Q1.p = (subclassof1 seq (Q1 or eps()) seq subclassof) or
        (type1 seq (Q1 or eps()) seq type)
```

```
val subclassof1: Symbol[L, N, _] = inE("subClassOf")
val subclassof: Symbol[L, N, _] = outE("subClassOf")
val type1: Symbol[L, N, _] = inE("type")
val _type: Symbol[L, N, _] = outE("type")
val grammar: Symbol[L, N, _] =
  syn((subclassof1 ~ syn(grammar | ε) ~ subclassof) |
    (type1 ~ syn(grammar | ε) ~ _type))
```

Запросы для исследования времени (yago)

```
(v { it.properties[”id”] == ”40324616” } seq
(inE { it.label == ”P92580544” } seq inV()).some seq
(inE { it.label == ”P13305537” } seq inV()).some seq
inE { it.label == ”P59561600” } seq inV() seq
inE { it.label == ”P74636308” } seq inV())
```

```
syn(V((e: Entity) => e.getProperty(”id”) == ”40324616”) ~
inE((e: Entity) => e.label() == ”P92580544”).+ ~
inE((e: Entity) => e.label() == ”P13305537”).+ ~
inE((e: Entity) => e.label() == ”P59561600”) ~
inE((e: Entity) => e.label() == ”P74636308”))
```