Parser-Combinators for Contex-Free Path Querying

Sophia Smolina
Electrotechnical University
St. Petersburg, Russia
sofysmol@gmail.com

Ekaterina Verbitskaia
Saint Petersburg State University
St. Petersburg, Russia
webmaster@marysville-ohio.com

Ilya Kirillov
Saint Petersburg State University
St. Petersburg, Russia
larst@affiliation.org

Semyon Grigorev
Saint Petersburg State University
St. Petersburg, Russia
semen.grigorev@jetbrains.com

ABSTRACT

Transparent intergration of domain-specific languages for graph-structured data access into general-purpose programming languages is an importatn for data-centric application development simplification. It is necessary to provide safety too (static errors checking, type chacking, etc) One of type of navigational queryes is a contex-free path queryes which stands more and m/home/ilya/-Downloads/graph (2).pdfore popular Context-free path querying reuquired in some areas, theoretical research, but no languages (cfS-PARQL) Grammars cpecification languages — combinators. We propose to use parser combinators technique to implement context-free path qurying We demonstrate library and show that it is applicable for realistic problems.

Abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, Abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, Abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, Abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, abstract,

CCS CONCEPTS

• **Information systems → Graph-based database models**; **Query languages for non-relational engines**; • **Software and its engineering** → *Functional languages*; • **Theory of computation** → *Grammars and context-free languages*;

KEYWORDS

Graph data bases, Language-constrained path problem, Context-Free path querying, Parser Combinators, Domain Specific Language, Generalized LL, GLL, Neo4J, Scala

ACM Reference Format:
Sophia Smolina, Ekaterina Verbitskaia, Ilya Kirillov, and Semyon Grigorev. 2018. Parser-Combinators for Contex-Free Path Querying. In *Proceedings of Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) 2018 (GRADES-NDA'18).* ACM, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.475/123_4

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
GRADES-NDA'18, June 2018, Houston, Texas USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

1 INTRODUCTION

When you develop a data-centric application, you want to use general purpose programming language and have an transparent and native access to data sources. The naiv approach, named string-embedded DSLs, which means that you have a driver which can execute a string with query and return (posibly untyped) result have some serious problems. First of all, it is necessary to use special DSL which may require additional knolages from developer. Moreover, string-embedded languahe is a source of errors and vulnerabilities which static detection is very hard [?]. These leeds to creation such special techniques as Object Relationship Mapping (ORM) or Language Integrated Query (LINQ) [? ?] which solve some problems, but have some difficulties with flexibility, for example with query decomposition and subquery reusing.

Applications which use graph structured data is also have such problem. There is a number of special languages for graph traversing/querying, such as SPARQL [?], Cypher [1], Gremlin [?]. Different techniques such as ORM It is necessary to integrate these languages into gp programming languge. , data access, languages integration for graph-structured data (or graph DB) access. In this paper we propose solution for naural transperent integration which provide deep syntax integration and do not require complex syntax extensions

One of useful type of graph queries is language-constarined path queries [?]. In this work we are focused on context-free path queries (CFPQ) which use context-free languages for constrains specification and used in such areas as bioinformatics [?], static code analysis [? ? ? ?], RDF processing [?]. Note that not only reachability information may be useful. Such important tasks as paths extraction, queries debuggibg and result processing [?] require appropriate representation of query result. There are a lot of theoretical research and problem-specific solutions [? ? ? ? ? ? ?], but there is no languages for applications develoer.

cfSPARQL [?] — separated language

One of the natural way to specify any language is specify its formal grammar which can be done by using special DSL based, for example, on EBNF-like notation [?]. The classical alternative way

[1] Cypher languge web page: https://neo4j.com/developer/cypher-query-language/. Access date: 16.01.2018

is a pasrer combinators technique which provide !!!Ekaterina, we need your help!!.

Unfortunately, classical combinators are based on LL(k) top-down parsing technique and can not handle left recursive and umbigues grammars [?]. In [?] authors show that it is possible to use ideas of Generalized LL [?] (GLL) for creation parser combinators which can handle arbitrary context-free grammars. Meerkat [2] parser combinators library is bsed on [?] and provide Shared Packed Parse Forest [?] (SPPF) — compact representation of parsing result. Furthermore, there is a solution which use GLL for CFPQ [?] and it is showed that SPPF can be used is finite structural representation of query result even when set of paths is infinite.

On the other hand, an idea to use combinators for graph traversing proposed in [?], but presented solution provide approximated handling of cycles in input graph and have a problem with left-recursive grammars. Authors pointed out that described idea is very close to classical parser combinators technique, but supported language class and restrictions are not discussed.

In this paper we show how to compose these ideas and present parser combinators for CFPQ which can handle arbitrary context-free grammar and provide structural representation of result. We make the following contributions in this paper.

(1) We show that it is possible to create parser combinators for context-free path querying which supports both arbitrary contex-free grammars and arbitrary graphs and provide finite structural representation of query result.
(2) We provide the implementation of parser combinators library in Scala. This library provide integration with Neo4J graph data base. Source code available on GitHub:https://github.com/YaccConstructor/Meerkat.
(3) We perform an evaluation on realistic data. Also we compare performance of our library with other GLL-bsed CFPQ tool and with Trails library. As a result we conclude that our solution provide good expressivity and performance to be applied for real-world problems.

2 PARSER-COMBITATORS FOR PATH QUERYING

Parser-Combinators is a way to describe context-free grammar in terms of functions and operations on them. Parser is a function which takes some input and returns either Success with result of parsing or Error in a case of failure. Such parsers are composable which makes it easy to create new parsers from existing ones.

One possible solutions of solution to create queries using parser-combinators is a Trails [?]. But Trails strugles with left-recursion grammars like and also may stuck on some graphs with loops by not yielding some paths in result stream.

Our work based on Meerkat parser-combinators library which can hadnle left recursion and as result has Shared Packed Parse Forest SPPF [?] which is a graph stores all possible ways to parse given input. From that SPPF we can get everything we need to know about our paths.

[2] Meerkat project repository: https://github.com/meerkat-parser/Meerkat. Access date: 16.01.2018

But Meerkat was made to work on a linear input; so, we extend input for Meerkat from linear to the graph one. That allows us to get all possible paths in graphs which is described by grammar.

Let us introduce an example. Let's assume we have context-free grammar G presented on Fig. 1. It produces so-called same genration query for pair a and b. Its Meerkat representation is in Fig. 2

$$\begin{aligned} S &\rightarrow \quad a\,A\,b \mid M \\ M &\rightarrow a\,b \end{aligned}$$

Figure 1: Example grammar

```
val S = syn("a" ~ A ~ "b" | M)
val M = syn("a" ~ "b")
```

Figure 2: Same-generation query in terms of parser combinators

Let's closely take a look at it. For every nonterminal in our CF grammar we create a val of `Nonterminal` type. `syn` is a macro which creates new nonterminal and automaticaly assigns a name of our val to it. Inside a `syn` macro we've got a defenition of nonterminal. It using two combinators ~ and `|`. The first one ~ says that after edge described by left operand we would like to have adjacence edge described by right one. `|` is an altenative combinator which have lower priority then ~ and used to describes possible alternativesof paths description. A new combinators can be created using existing ones like for nonterminal on Fig. 2 which makes parser combinators a powerfull technique for describing a paths in graphs.

The following table shows the some combinators avalible in Meerkat

Combinator	Description
`a ~ b`	`a` and then `b`
`a | b`	`a` or `b`
`a.?`	`a` or nothing
`a.*`	zero or more of `a`
`a.+`	at least one `a`

Table 1: Meerkat combinators

One of the most excited combinators feature is that queries using combinators can be dynamicaly generated. Let's take a look at Fig 3. `sameGen` creates a asme generation query for every pair of brackets in `bs`. For example, for `bs = List(("(", ")"), ("[", "]"))` it would return a parser equvalent to

```
val G = "(" ~  G.? ~ ")" | "[" ~  G.? ~ "]"
```

Armed with that function, let us generate a parser equal to a one presented on Fig. 1 by calling `sameGen(List(("a", "b")))`

And let's fially parse graph from Fig. 4 and get all paths described by same-generation grammar from it.

To do it let us use meerkat function `parseGraphFromAllPositions(parser,graph)` which applies

```
def sameGen(bs: List[(Nonterminal, Nonterminal)]) =
  syn { bs
    .map { case (x, y) => x ~ sameGen(bs).? ~ y }
    .foldLeft(AlternationBuilder.empty)(_ | _)
  }
```

Figure 3: Same generations query generator

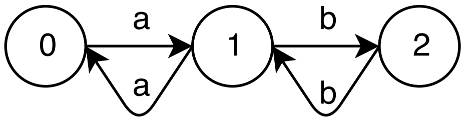


Figure 4: Example graph

given parser to given graph and gets from SPPF all pairs of nodes such that exists a path between them described by a parser.

The result for graph in Fig. 4 is $\{(1,0),(1,2),(0,0),(2,1),(2,2),(0,2),(0,1),(1,1)\}$, where (i,j) stands for the path from node with label i to the node with label j

Meerkat library consists of 3 main units (Fig. 5): the input data unit, the analysis unit, the visualization unit.

The input data unit can accept two main types: graphs and strings. It is necessary to implement the IGraph interface to analyze graphs with different implementations. There are two implementation of IGraph at the moment: Neo4JGraph and SimpleGraph. Neo4JGraph takes data from the Neo4j graph database. SimpleGraph allows you to describe the graph in the code of the program.

The analysis unit consists of parser and SPPFLookup. The parser gets the input data and the grammar, which is a input data query. The parser maps the input data to the input grammar and finds all the paths. The result is added to the SPPFLookup data structure, which contains all the paths in the input data corresponding to the grammar. All nodes in the SPPFLookup are in a single instance and re-used.

After the analysis is completed, the visualization unit gets SPPFLookup and SPPF query. The unit filters the path and then returns the file in dot format to the user.

3 EVALUATION

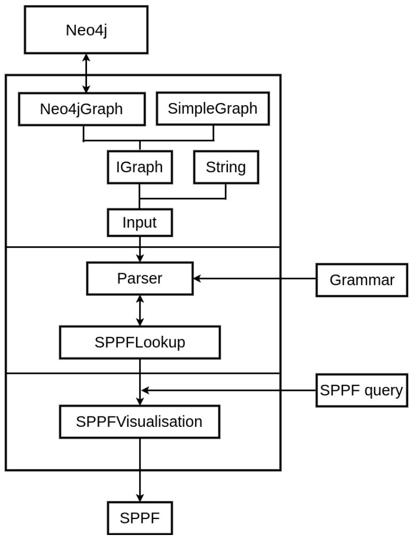
In this section we present evalution of Meerkat grph querying library. We show its perfomance on a classical ontology graphs for in memory graph and for Neo4j database, show application on may-alias static code analysis problem, and compare with Trails [?] library for graph traversals.

All tests are perfomed on a machine running Fedora 27 with quad-core Intel Core i7 2.5 GHz CPU with 8 GB of memory.

3.1 Ontology querying

One of well-known graph querying problems is a queries for ontologies [?]. We use Meerkat to evaluate it on some popular ontologies presented as RDF files from paper [?]. We convert

Figure 5: Architecture Meerkat



RDF files to a labeled directed graph like the following: for every RDF triple (*subject*, *predicate*, *object*) we create two edges (*subject*, *predicate*, *object*) and (*object*, $predicate^{-1}$, *subject*). On those graphs we apply two queries from the paper [?] which grammars are in Fig. 6, and Fig. 7

$$\begin{aligned}
&S \rightarrow subClassOf^{-1}\ S\ subClassOf\\
&S \rightarrow type^{-1}\ S\ type\\
&S \rightarrow subClassOf^{-1}\ subClassOf\\
&S \rightarrow type^{-1}\ type
\end{aligned}$$

Figure 6: Query 1 grammar

$$\begin{aligned}
&S \rightarrow\ B\ subClassOf\\
&B \rightarrow subClassOf^{-1}\ B\ subClassOf\\
&B \rightarrow subClassOf^{-1}\ subClassOf
\end{aligned}$$

Figure 7: Query 2 grammar

Meerkat represetaion of those can be easily created by using `sameGeneration` function with `(("subclassof-1", "subclassof"), ("type-1", "type"))` as brackets. Its direct form similar to its EBNF form and presnted in Fig. 8 and Fig. 9

The queries applied in two following ways.

- Convert RDF files to a graph input for meerkat and then directly parse on query 1 and query 2
- Convert RDF files to a Neo4j database and then parse this database on given queries

Ontology	#tripples	Query 1				Query 2			
		#results	In memory graph (ms)	DB query time (ms)	Trails (ms)	#results	In memory graph (ms)	DB query time (ms)	Trails (ms)
atom-primitive	425	15454	174	236	2849	122	49	56	453
biomedical-mesure-primitive	459	15156	328	398	3715	2871	36	52	60
foaf	631	4118	23	42	432	10	1	2	1
funding	1086	17634	151	175	367	1158	18	23	76
generations	273	2164	9	27	9	0	0	0	0
people_pets	640	9472	68	87	75	37	2	3	2
pizza	1980	56195	711	792	7764	1262	44	56	905
skos	252	810	4	29	6	1	0	1	0
travel	277	2499	23	93	34	63	2	2	1
univ-bench	293	2540	19	74	31	81	2	3	2
wine	1839	66572	578	736	3156	133	5	7	4

Table 2: Evaluation results for In Memory Graph and Graph DB

```
val S: Nonterminal = syn(
   "subclassof-1" ~ S ~ "subclassof" |
   "type-1" ~ S ~ "type" |
   "subclassof-1" ~ "subclassof" |
   "type-1" ~ "type")
```

Figure 8: Meerkat representation of Query 1

```
val S: Nonterminal = syn(
   "subclassof-1" ~ S ~ "subclassof" |
   "subclassof")
```

Figure 9: Meerkat representation of Query 2

Table 2 shows experimental results of those two aproaches over the testing RDF files where *number of results* is a number of pairs of nodes (v_1, v_2) such that exists S-path from v_1 to v_2.

The perfomance is about 2 times slower than in [?] and shows the same results. If compare the perfomance of in memory graph querying and database querying, the second one is slower in about $2-4$ times.

3.2 Static code analysis

Alias analysis is one of the fundamental static analysis problems [?]. Alias analysis checks may-alias relations between code expressions and can be formulated as a Context-Free language (CFL) reachability problem [?]. In that case program represeted as Program Expression Graph (PEG) [?]. Verticies in PEG are program expressions and edges are relations between them. In a case of analysisng C source code there is two kind of edges **D**-edge and **A**-edge.

- Pointer dereference edge (**D**-edge). For each pointer deference $*e$ there is a directed D-edge from e to $*e$.
- Pointer assignment edge (**A**-edge). For each assignment $*e_1 = e_2$ there is a directed A-edge from e_2 to $*e_1$

Also, for the sake of simplicity, there are edges labeled by $\overline{D}$ and $\overline{A}$ which corresponds to reversed D-edge and A-edge, respectively.

$$M \rightarrow \overline{D}\, V\, D$$
$$V \rightarrow (M?\, \overline{A})^*\, M?\, (A\, M?)^*$$

Figure 10: Context-Free grammar for the may-alias problem

Program	Code Size (KLOC)	Count of aliases		Time (ms)
		M aliases	V aliases	
wc-5.0	0.5K	0	174	350
pr-5.0	1.7K	13	1131	532
ls-5.0	2.8K	52	5682	436
bzip2-1.0.6	5.8K	9	813	834
gzip-1.8	31K	120	4567	1585

Table 3: Running may-alias queries on Meerkat on some C open-source projects

The grammar for may-alias problem from [?] presented in Fig. 10. It consists of two nonterminals **M** and **V**. It allows us to make two kind of queries for each of nonterminals **M** and **V**.

- **M** production shows that two l-value expression are memory aliases i.e. may stands for the same memory location.
- **V** shows that two expression are value aliases i.e. may evaluate to the same pointer value.

We made **M** and **V** queries on the code some open-source C projects. The results are presented on the Table 3

```
val M: Nonterminal = syn("nd" ~ V ~ "d")
val V: Nonterminal = syn(
   syn(M.? ~~ "na").* ~ M.? ~ syn("a" ~ M.?).*
)
```

Figure 11: Meerkat representation of may-alias problem grammar

It may be usefull for tools development.

3.3 Comparison with Trails

Trails [?] is a Scala graph combinator library. It provides traversers for describing paths in graphs in terms of parser combinators and allows to get results as a stream (maybe infinite) of all possible paths described by composition of basic traversals. Trails as well as Meerkat support parsing in memory graphs, so we compare perfomance of Trails and Meerkat on the queries from subsection 3.1. Query 1 and Query 2 in terms of Trails are in Fig. 12 and Fig. 13.

Here queries made in same way as in Meerkat. `S` traversal returns pairs (*begin node*, *start node*) of S-path. Combine operators ~ and ~> in queries made in the way to get only last node from path.

```
val B = (out("type-1") ~> out("type")) |
   (out("subclassof-1") ~> B ~> out("subclassof")) |
   (out("type-1") ~> B ~> out("type")) |
   (out("subclassof-1") ~> out("subclassof"))
val S = V ~ B
```

Figure 12: Trails representation of Query 1

```
val B = out("subclassof") |
   (out("subclassof-1") ~> B ~> out("subclassof"))
val S = V ~ B
```

Figure 13: Trails representation of Query 2

The result of comparation are in table 2. Trails gives the same results as Meeerkat (column *results* in table 2) but slower than Meerkat.

4 CONCLUSION

We propose a native way to integrate language for language-constrained path querying into general purpose programming language. We implement it and show that our implementation can be applied for real problems. Arbitrary context-free grammars for querying.

Code is available on GitHub:

Future work is

Technical improvements: sppf as a set of paths, combinators for vertices information processing, etc

SPPF utilization for debugging and results processing

Attributed grammars processing to provide mechanim for semantics calcualtion, filtration, etc