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
kajigor@gmail.com

Ilya Kirillov
Saint Petersburg State University
St. Petersburg, Russia
kirillov.ilija@gmail.com

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

ABSTRACT

A transparent integration of a domain-specific language for specification of context-free path queries (CFPQs) into a general-purpose programming language as well as static checking of errors in queries may greatly simplify the development of applications utilizing CFPQs. Such techniques as LINQ and ORM can be used for the integration, but they have issues with flexibility: query decomposition and reusing of subqueries are a challenge. Adaptation of parser combinators technique for paths querying may solve these problems. Conventional parser combinators process linear input and only the Trails library is known to apply this technique for path querying. Trails suffers the common parser combinators issue: it does not support left-recursive grammars and also experiences problems in cycles handling. We demonstrate that it is possible to create general parser combinators for CFPQ which support arbitrary context-free grammars and arbitrary input graphs. We implement a library of such parser combinators and show that it is applicable for realistic tasks.

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 Context-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

1 INTRODUCTION

One useful type of graph queries is language-constarined path queries [3]. There are several languages for graph traversing/querying which support constraints formulated in terms of regular languages. For example SPARQL [18], Cypher [1], and Gremlin [22]. In this work we are focused on context-free path queries (CFPQ) which use context-free languages for constraints specification and are used in bioinformatics [24], static code analysis [4, 17, 20, 28], and RDF processing [27]. There are a lot of theoretical research and problem-specific solutions on CFPQ [2, 8, 9, 16, 21, 24, 26], but there is a single known graph query languege which supports CF constraints: cfSPARQL [27].

When one develops a data-centric application, one wants to use general purpose programming language and have a transparent and native access to data sources. String-embedded DSLs is one way to do it. It utilizes a driver to execute a query written as a string and to return a possibly untyped result. This approach has serious drawbacks. First of all, a DSL may require additional knowledge from a developer. Moreover, a string-embedded language itself is a source of possible errors and vulnerabilities static detection of which is very difficult [5]. In trying to solve these issues, such special techniques as Object Relationship Mapping (ORM) or Language Integrated Query (LINQ) [6, 15] were created. Unfortunately, they still experience difficulties with flexibility: for example with the query decomposition and the reusing of subqueries. In this paper, we propose a transparent and natural integration of CFPQs into a general-purpose language.

One natural way to specify a language is to specify its formal grammar which can be done by using special DSL based, for example, on EBNF-like notation [25]. The classical alternative way is a pasrer combinators technique which provide !!!Ekaterina, we need your help!!.

Unfortunately, classical combinators implement top-down parsing and cannot handle left recursive and ambiguous grammars [?]. In [11] authors demonstrate a set of parser combinators which can

---

---

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

handle arbitrary context-free grammars by using ideas of Generalized LL [23] (GLL). Meerkat [2] parser combinators library is based on [11]. The result of parsing is represented in a compact form as Shared Packed Parse Forest [19] (SPPF). Paths extraction, queries debugging and result processing [10] require an appropriate representation of the query result. It is showed that SPPF is a suitable finite structural representation of a CFPQ query result, even if the set of paths is infinite [7].

An idea to use combinators for graph traversing has already been proposed in [13], but the solution presented provides only approximated handling of cycles in the input graph and does not support left-recursive grammars. Authors pointed out that the idea described is very similar to the classical parser combinators technique, but the supported language class or restrictions are not discussed.

In this paper we show how to compose these ideas and present the parser combinators for CFPQ which can handle arbitrary context-free grammars and provide structural representation of the 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 work on both arbitrary contex-free grammars and arbitrary graphs and provide a finite structural representation of the query result.
(2) We provide the implementation of the parser combinators library in Scala. This library provides an integration to Neo4J graph data base. Source code is available on GitHub:https://github.com/YaccConstructor/Meerkat.
(3) We perform an evaluation on realistic data. Also we compare the performance of our library with another GLL-bsed CFPQ tool and with the Trails library. We conclude that our solution is expressive and performant enought to be applied to the real-world problems.

2 PARSER COMBITATORS FOR PATH QUERYING

Parser combinators provide a way to specify a language syntax in terms of functions and operations on them. A parser in this framework is usually a function which consumes a prefix of an input and returns either a parsing result or an error, if the input is erroneous. Parsers can be composed by using a set of parser combinators to form more complex parsers. A parser combinators library provides with a set of basic combinators (such as sequential application or choice), and there can also be user-defined combinators. Most parser combinators libraries, including the Meerkat library, can only process the linear input — strings or some kind of streams. We extend the Meerkat library to work on the graph input.

Meerkat library is a general parser combinators library; by using memoization, continuation passing style and the ideas of Johnson [12] it supports arbitrary context free specifications. This library is closely related to the Generalized LL algorithm and since GLL can be generalized for context free path querying [7], the adaptation of Meerkat is reasonable too [!!! Đ£ĐţŇĂĐţŇĎĐ¿ŇĂĐijŇČĐżĐÿŇĂĐ¿ĐšĐřŇĆŇŇ !!!]. It can be

done by using an odservation which for (string or graph) parsing we need only to provide function for getting symbols follower by specified position.

The combinators our library provides are presented in table 1. Basic parser combinators for mathching strings are implicitly generated whenever a string is used within a query. The classical same generation query [1] can be writen using the library as presented in Fig. 1.

Combinator	Description
a ~ b	sequentional parsing: a then b
a | b	choice: a or b
a.?	optional parsing: a or nothing
a.*	repetition of zero or more a
a.+	repetition of at least one a

Table 1: Meerkat combinators

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

Figure 1: The same generation query in Meerkat

The most exciting feature of our library is that queries can be used as first class values, which means greater generalization and comosition. Function `sameGen` presented in Fig 3 is a generalization of the same generation query which is independent from the environment such as the input graph structure or other parsers. It can be used for creation of different queries, including the one presented in Fig 1: it is the result of application of `sameGen` to the appropriate relations (which may be treated as "brackets"). Another application of the `sameGen` can be founded in Fig. 6.

```
val query1 = syn(sameGen(List(
    ("subclassof-1", "subclassof"),
    ("type-1", "type"))))
```

Figure 2: Query ?? as an application of `sameGen`

```
def sameGen(brs) =
  bs.map { case (lbr, rbr) =>
          lbr ~ syn(sameGen(bs).?) ~ rbr }
  match {
    case x :: Nil => syn(x)
    case x :: y :: xs =>
      syn(xs.foldLeft(x | y)(_ | _))
  }
```

Figure 3: Generic function for same generations query

Running a query over an input graph retrieves the list of pairs (i, j) where each pair corresponds to the set of paths from the node

i to the node *j*. Running the same generation query from Fig. 6 over the graph in Fig. 4 returns $\{(1, 0), (1, 2)\}$ as a result. Internally in Meerkat this paths represented as SPPF [19]. Simplified version of SPPF for this query preseted on Fig. 5 where rounded rectangles represents nonterminals, others rectangles represent productions. Every rectangle contains nontermonal name or production rule, start and end nodes of thr input graph. Gray rectangels are start nonterminals.
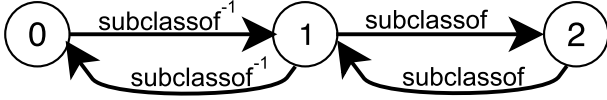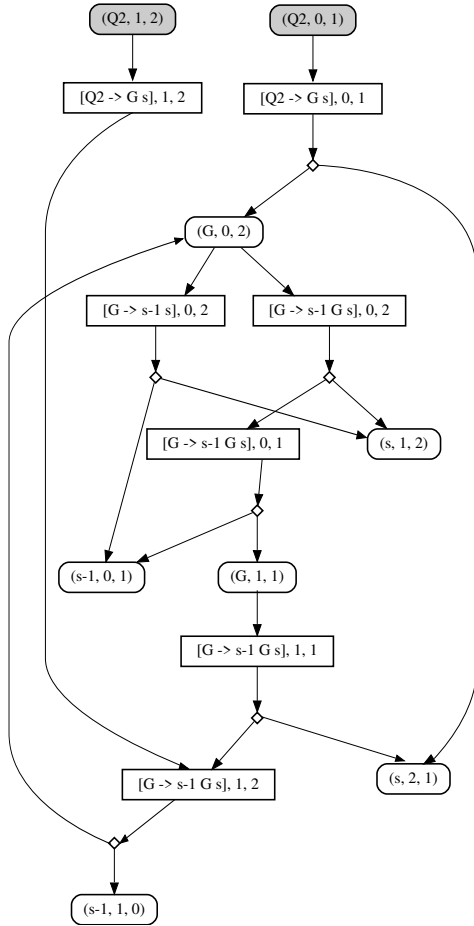

Figure 4: Example input graph.

Figure 5: SPPF: result of application the same generation query 6 to the graph 4

3 EVALUATION

In this section we present evolution of our graph querying library. We measure its performance on classical set of ontology graphs [27]:

both when the graph is loaded into the RAM and for the integration with the Neo4j database and compare it with the solution based on the GLL parsing algorithm [7] and the Trails [13] library for graph traversals. We also show how may-alias static code analysis can be done by means of the developed library.

All tests have been perfomed on a computer running Fedora 27 with quad-core Intel Core i7 2.5 GHz CPU and 8 GB of RAM.

3.1 Ontology querying

Querying for ontologies is a well-known graph querying problem. We evaluate our library on some popular ontoloies which are presented as RDF files in the paper [27]. Frist, we convert RDF files to a labeled directed graph in following manner: we create two edges $(subject, predicate, object)$ and $(object, predicate^{-1}, subject)$ for every RDF triple $(subject, predicate, object)$. Then the graph is either loaded into the Neo4j database or loaded directly into the memory. Then we run two queries from the paper [7] for these graphs. The grammars for the queries are presented in Fig. 2 and Fig. 6.

```
val query2 = syn(
  sameGen(List(("subclassof-1", "subclassof"))) ~
   "subclassof")
```

Figure 6: Query 2 grammar

The performace results are shown in the table 2. *#triples* reflects the size of the RDF file with the corresponding ontology, *#results* is a number of pairs of the nodes between which at least one S-path exists.

Meerkat-based and GLL-based [7] solutions show the same results (column *#results*) and the Query 1 runs up to two times faster on Meerkat-based solution than on GLL-based, meanwhile GLL-based solution is faster for the Query 2. Querying the database is naturally $2 - 4$ times slower than querying the graphs located in the RAM.

3.2 Static code analysis

Alias analysis is a fundamental static analysis problem [14]: it checks may-alias relations between code expressions and can be formulated as a Context-Free language (CFL) reachability problem [20] which is closely related to context free path querying problem. In this analysis a program is represeted as a Program Expression Graph (PEG) [28]. Verticies in a PEG correspond to program expressions and edges are relations between them. There are two types of edges possible while analysing C source code: **D**-edge and **A**-edge.

- Pointer dereference edge (**D**-edge). For each pointer dereference $*e$ 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 $*e_1$

For the sake of simplicity, we add edges labeled by $\overline{D}$ and $\overline{A}$ which correspond to the reversed **D**-edge and **A**-edge, respectively.

The grammar for may-alias problem from [28] is presented in Fig. 7. It contains two nonterminals **M** and **V** and allows us to make two kind of queries for each of them.

Ontology	#tripples	Query 1					Query 2				
		#results	In memory graph (ms)	DB query (ms)	Trails (ms)	GLL (ms)	#results	In memory graph (ms)	DB query (ms)	Trails (ms)	GLL (ms)
atom-primitive	425	15454	112	167	2849	232	122	49	52	453	19
biomedical-measure-primitive	459	15156	226	247	3715	482	2871	34	42	60	26
foaf	631	4118	16	25	432	29	10	1	2	1	1
funding	1086	17634	123	152	367	179	1158	18	23	76	13
generations	273	2164	6	21	9	12	0	0	0	0	0
people_pets	640	9472	63	84	75	80	37	2	3	2	1
pizza	1980	56195	544	650	7764	793	1262	44	47	905	50
skos	252	810	4	9	6	6	1	0	1	0	0
travel	277	2499	21	55	34	21	63	2	2	1	2
univ-bench	293	2540	15	43	31	24	81	2	2	2	1
wine	1839	66572	543	727	3156	606	133	5	7	4	5

Table 2: Comparation of Meerkat, Trails and GLL perfomance on ontologies

$$M \to \overline{D} \ V \ D$$

$$V \to (M? \ \overline{A})^* \ M? \ (A \ M?)^*$$

Figure 7: Context-Free grammar for the may-alias problem

Program	Code Size (KLOC)	Count of aliases		Time (ms)
		M aliases	V aliases	
wc-5.0	0.5	0	174	107
pr-5.0	1.7	13	1131	63
ls-5.0	2.8	52	5682	253
bzip2-1.0.6	5.8	9	813	71
gzip-1.8	31	120	4567	227

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

- **M** production means that two l-value expressions are memory aliases i.e. may refer to the same memory location.
- **V** production means that two expressions are value aliases i.e. may evaluate to the same pointer value.

We run the **M** and **V** queries on some open-source C projects. The results are presented in table 3

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

Figure 8: Meerkat representation of may-alias problem grammar 7

3.3 Comparison with Trails

Trails [13] is a Scala graph combinator library. It provides traversers for describing paths in graphs which resemble parser combinators and calculates possibly infinite stream of all possible paths described by the composition of basic traversers. Both Trails and Meerkat

support parsing of the graphs located in RAM, so we compare perfomance of Trails and Meerkat on the ontology queries described above: the results are presented in the table 2. Trails and Meerkat compute the same results, but Trails is up to 10 times slower than Meerkat.

To summarise, we showed that parser combinators are expressive enough to be used for implemening real queries. Our implementation is as performant as the other existing combinators library and is comparable with the GLL-based solution.

4 CONCLUSION

We propose a native way to integrate language for context-free path querying into general purpose programming language. Our solution can handle arbitrary context-free grammars and arbitrary graphs. Proposed approach is language-independent and may be implemented for closely all general-purpose programming languages. We implement it in Scala programming language and show that our implementation can be applied for real problems.

We can propose some possible directions of future work. Firs of all it is necessary to extend library with combinators for vertices information processing. The next technical improvement is crateion of user-friendly interface for SPPF processing. For example, SPPF can be presented as a set of paths with additional information about its structure. It may be useful for SPPF utilization for debugging and query result processing.

Another direction is semantic actions (or attributed grammars) handling. It is useful for specification user-defined actions, such as filters, over subqueres result, which can make queryes more expressive. It is impossible in general case, but some techniques such as lazy evaluations can help to provide a technically appropriate solution. An important theoretcal question is for which class of sematic actions it is possible to provide precise general solution.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. Foundations of Databases. (1995).

[2] Pablo Barceló, Gaelle Fontaine, and Anthony Widjaja Lin. 2013. Expressive Path Queries on Graphs with Data. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 71–85.

[3] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-language-constrained path problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.

[4] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification inference using context-free language reachability. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 553–566.

[5] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. 2018. String Analysis for Software Verification and Security. (2018).

[6] James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-integrated Query. *SIGPLAN Not.* 48, 9 (Sept. 2013), 403–416. https://doi.org/10.1145/2544174.2500586

[7] Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free Path Querying with Structural Representation of Result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3166094.3166104

[8] Jelle Hellings. 2014. Conjunctive context-free path queries. (2014).

[9] Jelle Hellings. 2015. Path Results for Context-free Grammar Queries on Graphs. *CoRR* abs/1502.02242 (2015). http://arxiv.org/abs/1502.02242

[10] Piotr Hofman and Wim Martens. 2015. Separability by short subsequences and subwords. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 31. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[11] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. 2016. Practical, General Parser Combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2847538.2847539

[12] Mark Johnson. 1995. Memoization in Top-down Parsing. *Comput. Linguist.* 21, 3 (Sept. 1995), 405–417. http://dl.acm.org/citation.cfm?id=216261.216269

[13] Daniel Kröni and Raphael Schweizer. 2013. Parsing Graphs: Applying Parser Combinators to Graph Traversals. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/2489837.2489844

[14] Thomas J. Marlowe, William G. Landi, Barbara G. Ryder, Jong-Deok Choi, Michael G. Burke, and Paul Carini. 1993. Pointer-induced Aliasing: A Clarification. *SIGPLAN Not.* 28, 9 (Sept. 1993), 67–70. https://doi.org/10.1145/165364.165387

[15] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 706–706. https://doi.org/10.1145/1142473.1142552

[16] A. Mendelzon and P. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Computing* 24, 6 (1995), 1235–1258.

[17] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2006. Existential label flow inference via CFL reachability. In *SAS*, Vol. 6. Springer, 88–106.

[18] Eric Prud, Andy Seaborne, et al. 2006. SPARQL query language for RDF. (2006).

[19] Joan Gerard Rekers. 1992. *Parser generation for interactive environments*. Ph.D. Dissertation. Universiteit van Amsterdam.

[20] Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS '97)*. MIT Press, Cambridge, MA, USA, 5–19. http://dl.acm.org/citation.cfm?id=271338.271343

[21] Juan L Reutter, Miguel Romero, and Moshe Y Vardi. 2015. Regular queries on graph databases. *Theory of Computing Systems* (2015), 1–53.

[22] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, 1–10.

[23] Elizabeth Scott and Adrian Johnstone. 2010. GLL parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189.

[24] Petteri Sevon and Lauri Eronen. 2008. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics* 5, 2 (2008), 100.

[25] Niklaus Wirth. 1996. Extended Backus-Naur Form (EBNF). *ISO/IEC* 14977 (1996), 2996.

[26] Mihalis Yannakakis. 1990. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 230–242.

[27] Xiaowang Zhang, Zhiyong Feng, Xin Wang, Guozheng Rao, and Wenrui Wu. 2016. Context-free path queries on RDF graphs. In *International Semantic Web Conference*. Springer, 632–648.

[28] Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/1328438.1328464