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,

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

*This work is supported by grant from JetBrains Research.

GRADES-NDA'18, June 2018, Houston, Texas USA

ACM ISBN 123-4567-24-567/08/06…$15.00
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 [17], Cypher [1], and Gremlin [21]. 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 [23], static code analysis [4, 16, 19, 27], and RDF processing [26]. There are a lot of theoretical research and problem-specific solutions on CFPQ [2, 8, 9, 15, 20, 23, 25], but there is a single known graph query languege which supports CF constraints: cfSPARQL [26].

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, 14] 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 [24]. 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 handle arbitrary context-free grammars by using ideas of Generalized LL [22] (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 [18] (SPPF). Paths extraction, queries

[1] Cypher languge web page: https://neo4j.com/developer/cypher-query-language/. Access date: 16.01.2018
[2] Meerkat project repository: https://github.com/meerkat-parser/Meerkat. Access date: 16.01.2018

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 alreadt been proposed in [12], 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 the 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. 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 ideas of [!!!ÑĆÑČÑĆ Đ¡ÑČĐűĐ¡Đř ŇĄŇĄÑŃĐžĐžĐř Đ¡Đř johnson!!!] 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 [!!! ĐŠĐ¿ÑĆ ŇĆÑČÑĆ Đ¡ĐřĐťĐ¿ Đ£Đ¿ĐťÑČĐijĐřÑĆÑŇ !!!].

Let us introduce an example of the same generation query by using Meerkat. Context-free grammar G presented in Fig. 1 and its Meerkat representation is in Fig. 2

Let us take a look at it. For every nonterminal in our CF grammar we create a val of `Nonterminal` type. Strings are implicitly converted to terminals. `syn` is a macro which creates new nonterminal and

$$S \rightarrow subClassOf^{-1}\ S\ subClassOf$$
$$S \rightarrow type^{-1}\ S\ type$$
$$S \rightarrow subClassOf^{-1}\ subClassOf$$
$$S \rightarrow type^{-1}\ type$$

Figure 1: Query 1 grammar

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

Figure 2: Meerkat representation of Query 1

automaticaly assigns a name of our val to it. Inside a `syn` macro we have got a defenition of nonterminal. It uses two combinators `~` and `|`. The first one `~` states 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 alternatives of paths description.

The following table shows some combinators available 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

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.

One of the most excited combinators feature is that queries using combinators can be used as first class values, which provide high ability for queryes generalization and comosition. Let's take a look at Fig 4. `sameGen` creates a same generation query for every pair of brackets in `brs`. This generaization is independent from environment (graph and other parsers), and we can use it for creation of different queries. For example, the application of `sameGen` which one can see in Fig. 3 builds a query which is equivalents to the query presented in Fig. 2.

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

Figure 3: Application of `sameGen` for buildin the query 1

```
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 4: Generic function for same generations query building

And let us finally parse graph from Fig. 5 and get all paths described by same-generation grammar from it.

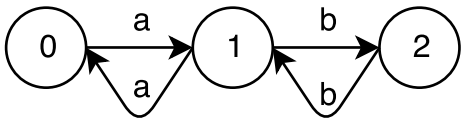

Figure 5: Example graph

To do it let us use meerkat function `parseGraphFromAllPositions(parser,graph)` which applies 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. 5 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. **??**): 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 [12] 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 [1]. We use Meerkat to evaluate it on some popular ontologies presented as RDF files from paper [26]. We convert 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 [7] which grammars are in Fig. 3, and Fig. 6

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

Figure 6: Query 2 grammar

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

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 [7] 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 [13]. Alias analysis checks may-alias relations between code expressions and can be formulated as a Context-Free language (CFL) reachability problem [19]. In that case program represeted as Program Expression Graph (PEG) [27]. 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.

The grammar for may-alias problem from [27] presented in Fig. 7. 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.

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

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

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	0.5K	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

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

3.3 Comparison with Trails

Trails [12] 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 ontology queries which are described above. The result of comparation are in table 2. Trails gives the same results as Meeerkat (column *results* in table 2) but slower than Meerkat.

To summarise we show that parser combinators are expressive enough for a formulation of real queries. Performance of our implementation, which is based on combinators for linear input parsing, is comparable with other similar solutions and enough for real-world problems 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 subqueryes 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] 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

[13] 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

[14] 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

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

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

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

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

[19] 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

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

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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