Parser-Combinators for Context-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 Databases, Language-Constrained Path Problem, Context-Free Path Querying, Parser Combinators, 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, 8 pages. https://doi.org/10.475/123_4

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-constrained path queries [2] which mean that constraints are formulated by using formal languages formalisms such as grammars. For example, suppuse that one have a graph presented in fig. 2 and want to find all paths which have a form $a^n b^n, n > 0$. It is possible to specify this constraint in terms of context-free grammar G, which is presented in fig 1.

$$S \to a\ S\ b$$
$$S \to a\ b$$

Figure 1: Context-free grammar G for the language $L = \{a^n b^n | n > 0\}$

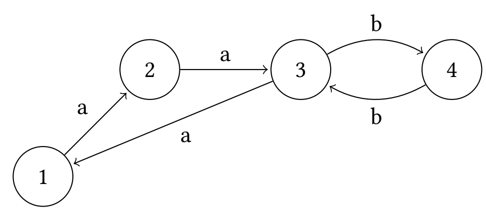

Figure 2: An example of input graph

There are several languages for graph traversing/querying which support constraints formulated in terms of regular languages. For example SPARQL [24], Cypher [1], and Gremlin [29]. In this work, we are focused on context-free path queries (CFPQs) which use context-free languages for constraints specification and are used in bioinformatics [33], static code analysis [3, 23, 26, 36, 40], and RDF processing [39]. There are a lot of problem-specific solutions and theoretical research on CFPQs [1, 7–9, 20, 28, 33, 37]. Among them, cfSPARQL [39] is a single known graph query language to support CF constraints. Generic solution for the integration of CFPQs into general-purpose languages is not discussed enough.

When one develops a data-centric application, one wants to use a general purpose programming language and have a transparent

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

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 [4]. In trying to solve these issues, such special techniques as Object Relationship Mapping (ORM) or Language Integrated Query (LINQ) [5, 15, 19] 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.

Note that CFPQs is applicable not only for graph data base querying but also for static code analysis (with name *context-free language reachability framework* or *IFDS framework*). CFPQ and context-free language reachability (CFL-reachability) is a different names of one thing: the first used mostely in database community and the second one in static code analysis. In 1995 Thomas Reps shows that the wide range of static code analysis problems can be formulated in terms of CFL-reachability in graph [26, 27]. This framework is widely used for solving different problems [6, 10, 23, 34, 36, 40] and in domain specific languages, like the Flix [17]. For softaware development it is more more convinient way to have an generic integrated framework (library) for general-purpose prgramming language to avoid integration problems. Our solurion may be use as a core of such framework because provide generic (problem and domain independent) mechanism for CFPQs evaluation.

It is necessary to find an appropriate technique for integration of context-free language specification into general-purpose programming languages. One natural way to specify a language is to specify its formal grammar which can be done by using a special DSL based, for example, on EBNF-like notation [35]. The classical alternative way is parser combinators [11] which provide all required features, including transparent integration, compile-time checks of correctness, high-level techniques for generalization.

An idea to use combinators for graph traversing has already been proposed in [14], 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, but the language class supported or restrictions are not discussed. This point is very important, because conventional combinators implement top-down parsing and cannot handle left-recursive and ambiguous grammars.

In [12], authors demonstrate a set of parser combinators which can handle arbitrary context-free grammars by using ideas of Generalized LL [30] (GLL). Meerkat [2] parser combinators library is based on [12] and provides result of parsing in a compact form as Shared Packed Parse Forest [25] (SPPF). 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], which can be used for paths extraction, queries debugging and processing of result.

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

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 a 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 context-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 [3] graph database. The 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-based CFPQ tool and with the Trails library. We conclude that our solution is expressive and performant enough to be applied to the real-world problems.

This paper is orgatized as follows. In section!!! we !!!! and in section !!! sdfsdf Then in section !!! we present a set of examples Evaluation Finally in section !!! we discuss possible directions of further research.

2 GENERALIZED PARSER COMBINATORS

In [14] showed that combinators techniques is applicable for graph travesing, but well-known problem of combinator with left recursion is not solved. But this problew was solved in classical parser-combinators [12] and implemented in Meerkat library. Meerkat library is a general parser combinators library implemented in Scala programming language; by using memoization, continuation-passing style and the ideas of Johnson [13], it supports arbitrary (left-recursive and ambigues) context-free specifications. This library implements classical set of combinators, supports action code specification, and provide `syn` macro for custom handling of recursive nonterminal description. Meercat creates compressed representation of parse forest (SPPF) which is useful for CFPQs results representation [7]. Time and space complexity of this soluton are cubic in worst case.

An example Meerkat specification of grammar 1 is presented in 3.

```
val S = syn("a" ~ S.? ~ "b")
```

Figure 3: Meercat specification of grammar 1

In [7] showed that generalized LL parsing algorithm [30] can be generalized to CFPQs processing and this generalization prodices efficient solution which can provide structured represenataion of query result. As the Meerkat library is closely related to the Generalized LL algorithm and since GLL can be generalized for context-free path querying, it is also possible to adapt Meerkat library for graph querying. It can be done by providing a function for retrieving the symbols which follow the specified position and utilizing it in the basic set of combinators. Details described below.

[3]Neo4j graph database site: https://neo4j.com/. Access date: 16.01.2018

2.1 SPPF

Structural representation. Derevation tree. Forest for unambiguous grammars. For graph too. Shared Packed Parse Forest (SPPF) [25] structure, description, usability for CFPQ.

Binarized Shared Packed Parse Forest (SPPF) [32] compresses derivation trees optimally reusing common nodes and subtrees. Version of GLL which utilizes this structure for parsing forest representation achieves worst-case cubic space complexity [31].

Binarized SPPF can be represented as a graph in which each node has one of four types described below. We denote the start and the end positions of substring as i and j respectively, and we call tuple (i, j) an *extension* of a node.

- **Terminal node** with label (i, T, j).
- **Nonterminal node** with label (i, N, j). This node denotes that there is at least one derivation for substring $\alpha = \omega[i..j-1]$ such that $N \Rightarrow^*_G \alpha, \alpha = \omega[i..j-1]$. All derivation trees for the given substring and nonterminal can be extracted from SPPF by left-to-right top-down graph traversal started from respective node.
- **Intermediate node**: a special kind of node used for binarization of SPPF. These nodes are labeled with (i, t, j), where t is a grammar slot.
- **Packed node** with label $(N \to \alpha, k)$. Subgraph with "root" in such node is one possible derivation from nonterminal N in case when the parent is a nonterminal node labeled with $(\Leftrightarrow (i, N, j))$.

An example of SPPF is presented in figure **??**.

3 PARSER COMBINATORS 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.

Extension is based on some common ideas.

- Intersection of context-free language and regular one is a context-free language and there are different constructive proves of this fact. Proposed solution is a yet another constructive prove and SPPF is a just user-friendly representation of result context-free grammar.
- Linear input is a simple case of graph: positions are vertices and tokens are edges labels. Each edge is going from position (vertex) i to position (vertex) $i + 1$.
- Parser can move pointer in input from position i to position $i + 1$ and create new state when token between i to position $i + 1$ matches with required in grammar. In case of graph processing there are more then one ways to go from current vertex i and it is possible to get more then one new state. Generalized parsing is designed to optimally handle steps which produce multiple new states and can handle help to handle this situation.
- We can treat the fact that token in input matches with required token from grammar as a predicate. This observation may be generalized: we can pass through edge if its label satisfies some predicate. This way we can flexibly handle labels of arbitrary types.
- Vertex may be converted to edge: all incoming edges are convert to oncoming into source of new edge, all outgoing are convert to outgoing from target of new edge. This way we can handle vertex and edge labelled graphs as edge labelled.

3.1 Library Structure

Querying process in our library consists of two subprocesses listed below.

- Applying a query to the graph and representation it as SPPF
- Applying semantic actions to SPPF which will allow us to retrieve all information we need from our SPPF. The specification we need to retrieve from SPPF also described in a terms of combinators using mapping combinators `^` and `&`. In details semantic action execution is described in section .

There are two main types of queries.

- Graph's edge and vertex is two main building blocks of queries. It is represented as a `Edge[N]` and `Vertex[L]` type where `N` and `L` is a type parameters of edge and vertex type, respectively. In a terms of CF-grammar edges and vertices are terminals. There are two combinators `E[N]` and `V[L]` for working with basic primitives described in section 3.2.
- A complete query has a `Query[L, N]` type. Query is a nonterminal in a terms of CF-grammars. For transforming an arbitrary query to a Query we have a `syn` macro which also gives it the same name as corresponding value. Later this naming will be used in SPPF.

3.2 The Set of Combinators

First we introduce a small example graph which represent a map and presented in fig. 4. Here we have some cities which can have road between them and this relation is shown in graph as an edge with label *road_to*. Each city labeled with name and with a country it belongs to.

And let we try to extract some information from this map.

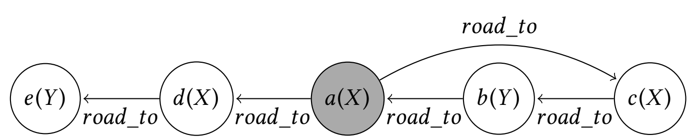

Figure 4: Example Input Graph of Roads. Vertex labels in a form of "city-name (country-name)"

First of all, for creating queries we need to work with edges and vertices. There are two main functions for that:

- `V[L](predicate: L =>Boolean)` combinator for working with vertices. Accepts a predicate and parses only vertices which satisfies that predicate
- `E[N](predicate: N =>Boolean)` combinator for working with edges. Accepts a predicate and parses only edges which satisfies that predicate

Suppose that we would like to select cities from our graph which belongs to some country. For that we use function `V`: `V[L]((e: Entity) =>e.country ="County_Name")`. Here `Entity` is a property container for graph entities: edges and vertices. All properties usings (like `(e: Entity).country`) are converted to the corresponding graph entity properties using Scala's `Dynamic` trait. Also, for the sake of simplicity, we will not explicitly specify `Entity` type for predicates. Now let us build a query which gets all roads from city in country X to the city in country Y. For that we can use a sequentialital combinator `~`. It allows to create queries which sequentialy applies two queries one after another. When we have subquery for retrieving a vertex with specific city, let's call it `city(name: String)` and a subquery `roadTo` for retrieving road edges. Let us finally build a query `city("X") ~ roadTo ~ city("Y")` which will give us requested set of paths from our graph. The full query with subqueries is shown on fig. 6.

Now we would like to get all pair of cities which have a road between them. So we need to transform our query to use semantic actions which is described in 3.4 section. Now let us specify what we want from every our query. From the `city` query we want only city name, so we need to map a result of basic vertex combinator. For that case we have a `^` combinator we can write `def city(name: String) =syn(V(e.value() ==name) ^ (_.value))` to achive that. In `ourPath` query we need first and second cities to be represented as a pair. For that we have a `&` combinator which will map our sequence to a pair of strings. The final representation is shown on 7. Now when we execute that query we will get a list which consists of all pairs of city's names which have a road between.

The whole set of basic combinators our library provides are presented in table 1. It consists of two kind of combinators. The first kind creates new parsers from existing ones, meanwhile the second one allows mapping parsers result. Parsers for matching strings are implicitly generated whenever a string is used within a query.

Combinator	Description
`a ~ b`	sequential 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`
`a ^ f`	apply `f` function to `a` if `a` is a token
`a ^^`	capture output of `a` if `a` is a token
`a & f`	apply `f` function to `a` if `a` is a parser
`a &&`	capture output of `a` if `a` is a parser

Table 1: Meerkat combinators

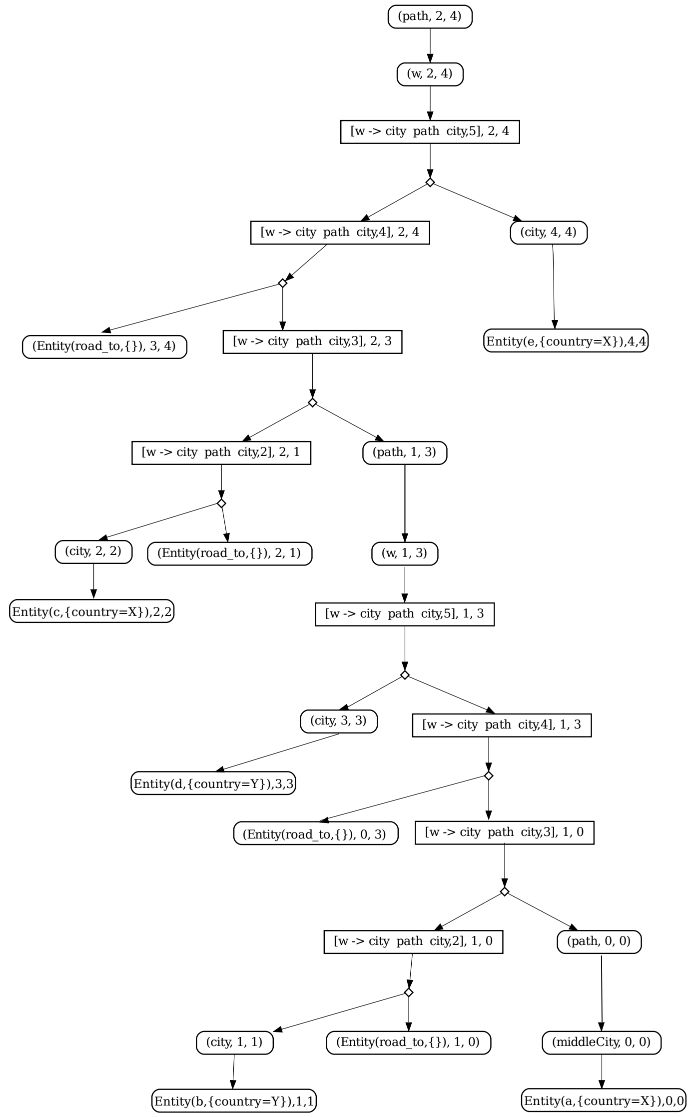

Figure 5: SPPF: result of applying cities query to the graph 4

3.3 Generic interface for input

Combinators is a generic way to describe a query and when we have a query we want to execute that query on some graph considering it as an input for our query. The cool thing is that query

```
def city(country: String) =
  V(e.country == country)
val roadTo = E(_.value() == "road_to")
val ourPath =
  city("X") ~ roadTo ~ city("Y")
```

Figure 6: Path query

```
def city(country: String) =
  syn(V(e.country() == country) ^ (_.name))
val roadTo = E(_.value() == "road_to")
val ourPath =
  syn(city("X") ~ roadTo ~ city("Y") &
    { case c0 ~ c1 => (c1, c2) }
```

Figure 7: Path query

execution mechanism may be fully separated from graph representation. We need only to have access to two very low-level functions, one for working with edges and one for vertices. The first one would allow to get all edges outcoming from current vertex and also satisfies given predicate. The second one will allow to check if current vertex satisfies given predicate. That interface is presented on fig .8. It has two type parameters: `L` for edge labels and `N` for nodes. We have implementation of that input for the next data sources:

- Neo4jInput — input source for working with graph database Neo4J;
- GraphxInput — input source for working with graph presented in memory using GraphX library;
- LinearInput — input source for working with linear input data like strings.

```
trait Input[+L, +N] {
  def filterEdges(nodeId: Int,
      predicate: L => Boolean): Seq[(L, Int)]
  def checkNode(nodeId: Int,
      predicate: N => Boolean): Option[N]
}
```

Figure 8: Generalized input interface

As far as required functions is very simple, we hope that this interface can be implemented for arbitrary storage of graph-structured data. Note, that currently we use `Int` as unique identifier for nodes (the `nodeId` parameter). It may be a technical restriction by the next two reason.

- It is impossible to use our library for correct processing of graph with more then `MAX_INT` nodes).
- It is necessary to provide such identifiers. Many systems use unique identifiers by default, but in some cases it may be necessary to implement required functionality manually.

3.4 Semantic Actions

Each path query produces a parse result stored in SPPF. This representation is very rich but hard to use and understand. That is why our library provides a mechanism which allows you to extract and process any useful data stored in parse result. This mechanism is called semantic actions. In general, they give you an opportunity to apply any function to parsed token or sequence. Now, let's understand how actions can be used in queries and how they are implemented in our library.

There are two main semantic action binders `^` and `&`. First of them is used when we need to perform some action on primitive tokens such as vertices or edges.

```
// Defined in Terminal[+L] (edge) parser
def ^[U](f: L => U) =
  new SymbolWithAction[L, Nothing, U] {...

// Defined in Vertex[+N] parser
def ^[U](f: N => U) =
  new SymbolWithAction[Nothing, N, U] {...
```

Second is used when we need to process a result of combination of parsers.

```
// Defined in Symbol[+L, +N, +V] parser
def &[U](f: V => U) =
  new SymbolWithAction[L, N, U] {...
```

But actually, they both have the same behaviour, they produce a new parser that has the same parsing possibilities as an original parser but also have a binded function. Then, every SPPF node that will be produced by parser with binded function will have a reference to this function too.

So, these operations in composition with other combinators provides an instrument for data processing on which most queries are based. For example, `^` can extract some data from tokens and `&` applied to sequence of tokens can collect and process data returned by terminal parsers.

The main idea of execution of semantic actions remained the same as in the original Meerkat library excepting one aspect. For each node we still just execute all actions of its children, collect results and pass them as argument to current function. But what should executor do if SPPF has ambiguous nodes? Previous implementation just throws an exception in that case and it is reasonable because original library is written for linear parsing and most grammars allows disambiguation in that case.

However, even unambiguous grammar can produce ambiguous derivations during parsing of graphs. That's why we provide a feature that makes it possible to extract "all" trees stored in SPPF. The number of path deriving from given grammar can be infinite, for example, when graph has cycles. This reason we can provide only a lazy stream of trees that allows to take as much of them as you need. Our solution is based on breadth first search that yields an unambiguous SPPF corresponding to some derivation immediately after it was found.

Composition of trees extraction and semantic action execution gives us a function that we called `executeQuery`. It parses graph from all positions producing a list of SPPF roots, then it extracts all derivations from all roots, executes semantic actions and returns a stream of results.

4 EVALUATION

In this section, we present an evaluation of our graph querying library. We measure its performance on a classical set of ontology graphs [39]: 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 [14] 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 performed on a computer running Fedora 27 with quad-core Intel Core i7 2.5 GHz CPU and 8 GB of RAM.

4.1 Ontology querying

Querying for ontologies is a well-known graph querying problem. We evaluate our library on some popular ontologies which are presented as RDF files in the paper [39]. First, we convert RDF files to a labelled directed graph in the 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 is 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. **??** and Fig. **??**.

The performance results are shown in the table 2 where *#results* is a number of pairs of the nodes between which at least one S-path exists.

The Meerkat-based and the GLL-based [7] solutions show the same results (column *#results*) and the Query 1 runs up to two times faster on the Meerkat-based solution than on the GLL-based, meanwhile the 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.

4.2 Static code analysis

Alias analysis is a fundamental static analysis problem [18]: it checks may-alias relations between code expressions and can be formulated as a context-free language (CFL) reachability problem [26] which is closely related to context-free path querying problem. In this analysis, a program is represented as a Program Expression Graph (PEG) [40]. Vertices in a PEG correspond to program expressions and edges are relations between them. There are two types of edges possible while analyzing 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 labelled by $\overline{D}$ and $\overline{A}$ which correspond to the reversed **D**-edge and **A**-edge, respectively.

The grammar for may-alias problem from [40] is presented in Fig. 9. It contains two nonterminals **M** and **V** and allows us to make two kinds of queries for each of them.

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

Figure 9: Context-Free grammar for the may-alias problem

- **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 in table 3. We can conclude that our solution is expressive and performant enough to be used for static analysis problems which can be expressed as CFPQs.

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

Figure 10: Meerkat representation of may-alias problem grammar

4.3 Comparison with Trails

Trails [14] 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-based solution support parsing of the graphs located in RAM, so we compare the performance of Trails and Meerkat-based solution on the ontology queries described above: the results are presented in table 2. Trails and Meerkat-based solution compute the same results, but Trails is up to 10 times slower than Meerkat-based solution.

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

5 DISCUSSION AND CONCLUSION

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

We can propose some possible directions for the future work. First of all, it is necessary to formulate the creation of a user-friendly interface for SPPF processing. We can just extract reachability information, but SPPF contains mach more useful information. One such representation may be a set of paths with additional information about their structure. This may simplify debugging and query result processing.

Ontology	#nodes	#edges	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	291	685	15454	64	67	2849	232	122	29	79	453	19
biomedical-measure-primitive	341	711	15156	112	108	3715	482	2871	18	18	60	26
foaf	256	815	4118	11	11	432	29	10	1	1	1	1
funding	778	1480	17634	69	68	367	179	1158	9	9	76	13
generations	129	351	2164	4	4	9	12	0	1	0	0	0
people_pets	337	834	9472	37	37	75	80	37	1	1	2	1
pizza	671	2604	56195	333	325	7764	793	1262	17	18	905	50
skos	144	323	810	1	2	6	6	1	1	0	0	0
travel	131	397	2499	11	13	34	21	63	1	1	1	2
univ-bench	179	413	2540	10	10	31	24	81	1	1	2	1
wine	733	2450	66572	405	401	3156	606	133	2	3	4	5

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

Program	#edges	#nodes	Code Size (KLOC)	In memory graph (ms)	Neo4j graph (ms)	Trails graph (ms)	M aliases	V aliases
wc-5.0	332	770	0.5	0	2	3	174	107
pr-5.0	815	2062	1.7	11	12	14	1131	63
ls-5.0	1687	4734	2.8	43	51	170	5682	253
bzip2-1.0.6	632	1508	5.8	8	13	21	813	71
gzip-1.8	2687	7510	31	111	120	537	4567	227

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

In order to improve performance and investigate scalability of proposer solution it is necessary to try to implement parallel single machne and distributed GLL. It is not only algorithmic problem: to get practical solution we should choose appropriate tools, libraryes for parallel and distributet computing for Scala.

Another direction is a semantic actions computation, otherwise known as attributed grammars handling. It increases the expressiveness of queries by means of the specification of user-defined actions, such as filters, over subqueries result. Although it is impossible in general, techniques such as lazy evaluation can provide a technically adequate solution. Anoter possible direction is utilizatuin of relational programming (minikanren) which is aimed to search [?]. For what class of semantic actions it is possible to provide a precise general solution is a theoretical question to be answered.

Some important problems in static code analysis reqwuire languages more expressive than contex-free one. For example, context-sensitive data-dependence analysis may be precisely expressed in terms of linear-conjunctive language [22] reachability, but not context-free [38]. While problem formulation is precise, it is possible to get only approximated solution, because emptyness problem for linear-conjunctive languages is undersidable. It would be an interesting task to support not only linear-cinjunctive grammars, but arbitrary conjunctive grammars [21] in te library and investigate nature of approximation. Finally it would be interesting to create a core for static analysis framework based on language reachcbility.

Improved version of OpenCypher [16], which is the one of the most popular graph query languages, provides context-free path querying mechanism. Detailed comparison with it may provide more information for direcion of future work.

ACKNOWLEDGMENTS

The research was supported by the Russian Science Foundation grant 18-11-00100 and a grant from JetBrains Research.

REFERENCES

[1] 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.

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

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

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

[5] 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

[6] Andrei Marian Dan, Manu Sridharan, Satish Chandra, Jean-Baptiste Jeannin, and Martin Vechev. 2017. Finding Fix Locations for CFL-Reachability Analyses via Minimum Cuts. In *International Conference on Computer Aided Verification*. Springer, 521–541.

[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] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 106–117.
[11] Graham Hutton and Erik Meijer. 1996. Monadic parser combinators. (1996).
[12] 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
[13] 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
[14] 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
[15] Kazumasa Kumamoto, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2015. A System for Querying RDF Data Using LINQ. In *Network-Based Information Systems (NBiS), 2015 18th International Conference on*. IEEE, 452–457.
[16] Tobias Lindaaker. 2017. OpenCypher Path Patterns (CIP2017-02-06 Path Patterns). (2017). https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc#153-compared-to-context-free-languages
[17] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096
[18] 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
[19] 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
[20] A. Mendelzon and P. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Computing* 24, 6 (1995), 1235–1258.
[21] Alexander Okhotin. 2001. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics* 6, 4 (2001), 519–535.
[22] Alexander Okhotin. 2003. On the Closure Properties of Linear Conjunctive Languages. *Theor. Comput. Sci.* 299, 1 (April 2003), 663–685. https://doi.org/10.1016/S0304-3975(02)00543-1
[23] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2006. Existential label flow inference via CFL reachability. In *SAS*, Vol. 6. Springer, 88–106.
[24] Eric Prud, Andy Seaborne, et al. 2006. SPARQL query language for RDF. (2006).
[25] Joan Gerard Rekers. 1992. *Parser generation for interactive environments*. Ph.D. Dissertation. Universiteit van Amsterdam.
[26] 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
[27] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462
[28] Juan L Reutter, Miguel Romero, and Moshe Y Vardi. 2015. Regular queries on graph databases. *Theory of Computing Systems* (2015), 1–53.
[29] 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.
[30] Elizabeth Scott and Adrian Johnstone. 2010. GLL parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189.
[31] Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Science of Computer Programming* 78, 10 (2013), 1828–1844.
[32] Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. 2007. BRNGLR: a cubic Tomita-style GLR parsing algorithm. *Acta informatica* 44, 6 (2007), 427–461.
[33] Petteri Sevon and Lauri Eronen. 2008. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics* 5, 2 (2008), 100.
[34] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744
[35] Niklaus Wirth. 1996. Extended Backus-Naur Form (EBNF). *ISO/IEC* 14977 (1996), 2996.
[36] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven Context-sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 155–165. https://doi.org/10.1145/2001420.2001440
[37] 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.
[38] Qirun Zhang and Zhendong Su. 2017. Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/3009837.3009848
[39] 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.
[40] 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