# 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

Ilya Nozkin
Saint Petersburg State University
St. Petersburg, Russia
nozhkin.ii@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, Ilya Nozkin, and Semyon Grigorev. 2018. Parser-Combinators for Context-Free Path Querying. In *Proceedings of Ninth ACM SIGPLAN Symposium on Scala, 2018 (Scala 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn




*Scala 2018, September 2018, St. Louis, Missouri, United States*



ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn


## 1 INTRODUCTION

Graph querying is finding all paths in the graph which satisfy some constraints. If the constraints are specified with some formal language formalism, i.e. a grammar, it is called a language-constrained path query. For example, a grammar $S \rightarrow a\ S\ b \mid a\ b$ can be regarded as a query for the paths of the form $a^n b^n$, where $n \geq 1$. Querying the graph in the figure 1 returns the set of paths, each of which starts and ends in the vertex 3 and goes around the cycles in the graph the appropriate number of times: 3, 1, 2, 3, 1, 2, 3, 4, 3, 4, 3, 4, 3 and so on.

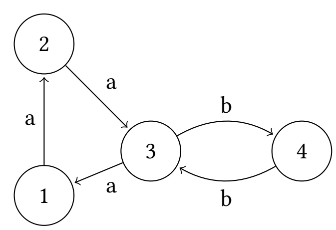


**Figure 1: An example of input graph**

Most existing graph traversing/querying languages, including SPARQL [21], Cypher [1], and Gremlin [26] support only regular languages as queries. For some applications regular languages are not expressive enough. Context-free path queries (CFPQ), on which we focus this paper, employ context-free languages for constraints specification. CFPQs are used in bioinformatics [30], static code analysis [2, 20, 23, 32, 36], and RDF processing [35]. Although there is a lot of problem-specific solutions and theoretical research on CFPQs [1, 6–8, 17, 25, 30, 33], cfSPARQL [35] is the 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 developing a data-centric application, one wants to use a general-purpose programming language and also to have a transparent and native access to data sources. One way to achieve this goal is to use string-embedded DSLs. In this approach, a query is written as a string, then passed on to a dedicated driver which executes it and returns a possibly untyped result. Despite the simplicity, string-embedded DSLs have serious drawbacks. First of all, they require the developer to learn the language itself, its features, runtime

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

and how the integration between the languages is implemented. DSLs are also a source of possible errors and vulnerabilities, static detection of which is a serious challenge [3]. Such techiques as the Object Relationship Mapping (ORM) or Language Integrated Query (LINQ) [4, 13, 16] partly solve these problems, but they still have issues with flexibility: both query decomposition and reusing of subqueries are a struggle. In this paper, we propose a transparent and natural integration of CFPQs into a general-purpose programming language.

Context-free path queries are known in various domains under different names. The *context-free language reachability framework* or *IFDS framework* is how it is called in the area of static code analysis. In [23, 24] Thomas Reps shows that the wide range of static code anlysis problems can be formulated in terms of CFL-reachability in the graph. This framework is used for such problems as the taint analysis [9], the alias analysis [31, 32, 36], the label flow analysis [20], and the fix locations problem [5]. What we propose in the paper can be used as a core of such framework since it provides both problem and domain independent mechanism for CFPQs evaluation.

We view parser combinators as the best way to integrate context-free language specifications into a general-purpose programming language. Parser combinators not only provide a transparent integration, but also compile-time checks of correctness and high-level techniques for generalization. An idea to use combinators for graph traversing has already been proposed in [12]. Unfortunately, the solution presented processes cycles in the input graph only approximately and is unable to handle left-recursive combinators, which is the most common issue of the approach. Authors pointed out that the idea described is similar to the classical parser combinators, but the language class supported or restrictions are not discussed.

Parser combinators are known to handle only a subset of context-free grammars: left recursion and ambiguity of the grammars are problematic. In [10], authors demonstrate a set of parser combinators which handles arbitrary context-free grammars by using ideas of the Generalized LL [27] algorithm (GLL). Meerkat [2] parser combinators library implements [10] and provides the parsing result in a compact form as the Shared Packed Parse Forest [22] (SPPF). SPPF is a suitable finite structural representation of a CFPQ result, even when the set of paths is infinite [6]. All the paths can be extracted from the SPPF—as the corresponding derivation trees—and further analysis can be done. It is also possible to run some further processing over the SPPF itself—without explicit paths extraction.

In this paper, we compose these ideas and present a set of parser-combinators for context-free path querying which handle arbitrary context-free grammars and provide a structural representation of the result. We make the following contributions in the paper.

(1) We show that it is possible to create a set of 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 implement 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 and 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.

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

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

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 CONTEX-FREE PATH QUERYING PROBLEM

In this section we formally describe the context-free path querying problem (or context-free reachability problem).

First, we introduce the necessary definitions.

- Context-free grammar is a quadruple $G = (N, \Sigma, P, S)$, where $N$ is a set of nonterminal symbols, $\Sigma$ is a set of terminal symbols, $S \in N$ is a start nonterminal, and $P$ is a set of productions.
- $\mathcal{L}(G)$ denotes a language specified by the grammar $G$, and is a set of terminal strings derived from the start nonterminal of $G$: $L(G) = \{\omega | S \Rightarrow^*_G \omega\}$.
- Directed graph is a triple $M = (V, E, L)$, where $V$ is a set of vertices, $L \subseteq \Sigma$ is a set of labels, and a set of edges $E \subseteq V \times L \times V$. We assume that there are no parallel edges with equal labels: for every $e_1 = (v_1, l_1, v_2) \in E, e_2 = (u_1, l_2, u_2) \in E$ if $v_1 = u_1$ and $v_2 = u_2$ then $l_1 \neq l_2$.
- $tag : E \to L$ is a helper function which returns a tag of a given edge.

$$tag(e = (v_1, l, v_2), e \in E) = l$$

- $\oplus : L^+ \times L^+ \to L^+$ denotes a tag concatenation operation.
- $\Omega$ is a helper function which constructs a string produced by the given path. For every $p$ path in $M$

$$\Omega(p = e_0, e_1, \ldots, e_{n-1}) = tag(e_0) \oplus \cdots \oplus tag(e_{n-1}).$$

We define the context-free language constrained path querying as, given a query in the form of a grammar $G$, to construct the set of the paths

$$Q(M, G) = \{p | p \text{ is path in } M, \Omega(p) \in \mathcal{L}(G)\}.$$

The CFL reachability problem is pretty similar and is formulated as follows:

$$Q(M, G) = \{(v_0, v_n) \mid \exists p \text{ is path in } M, p = v_0 \xrightarrow{l_0} \cdots \xrightarrow{l_{n-1}} v_n,$$

$$\Omega(p) \in \mathcal{L}(G))\}$$

.

Note that $Q(M, G)$ can be an infinite set, hence it cannot be represented explicitly. We show how to construct a compact data structure which stores all the elements of $Q(M, G)$ in a finite space; every path can be extracted from this representation.

## 3 GENERALIZED PARSER COMBINATORS

Combinators techniques are shown to be applicable for graph traversing [12], but it still suffers the common issue with left-recursive definitions. A general parser combinators library Meerkat [10], implemented in the Scala programming language, removes this restrction by using memoization, continuation-passing style, and the ideas of Juhnson [11]. It supports the arbitrary (left-recursive and ambiguous) context-free specifications, but it also supports the specification of action code, and provide a `syn` macro for custom handling of the recursive nonterminal descriptions. Meerkat constructs the compact representation of the parse forest in the form of SPPF, which can be used for CFPQs results representation [6]. The worst case time and space complexity of the solution is cubic.

A Meerkat specification of the language $\{a^n b^n \mid n \geq 1\}$ is presented in fig. 2.

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

**Figure 2: Meercat specification of** $\{a^n b^n \mid n \geq 1\}$

[6] showed that Generalized LL parsing algorithm [27] can be generalized to effectively process CFPQs and the query result can be finitely represented. 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 the 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.

### 3.1 SPPF

Structural representation. Derevation tree. Forest for unambiguous grammars. For graph too. Shared Packed Parse Forest (SPPF) [22] structure, description, usability for CFPQ.

Binarized Shared Packed Parse Forest (SPPF) [29] 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 [28].

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 **??**.

## 4 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.

### 4.1 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 4.2.

- A complete query has a `Query[L, N]` type. Query is a non-terminal 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.

## 4.2 The Set of Combinators

First we introduce a small example graph which represent a map and presented in fig. 3. 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.

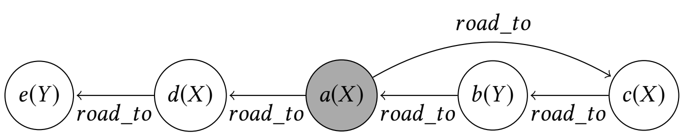


**Figure 3: Example Input Graph of Roads. Vertex labels in a form of "city-name (country-name)"**

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. 5.

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 4.4 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

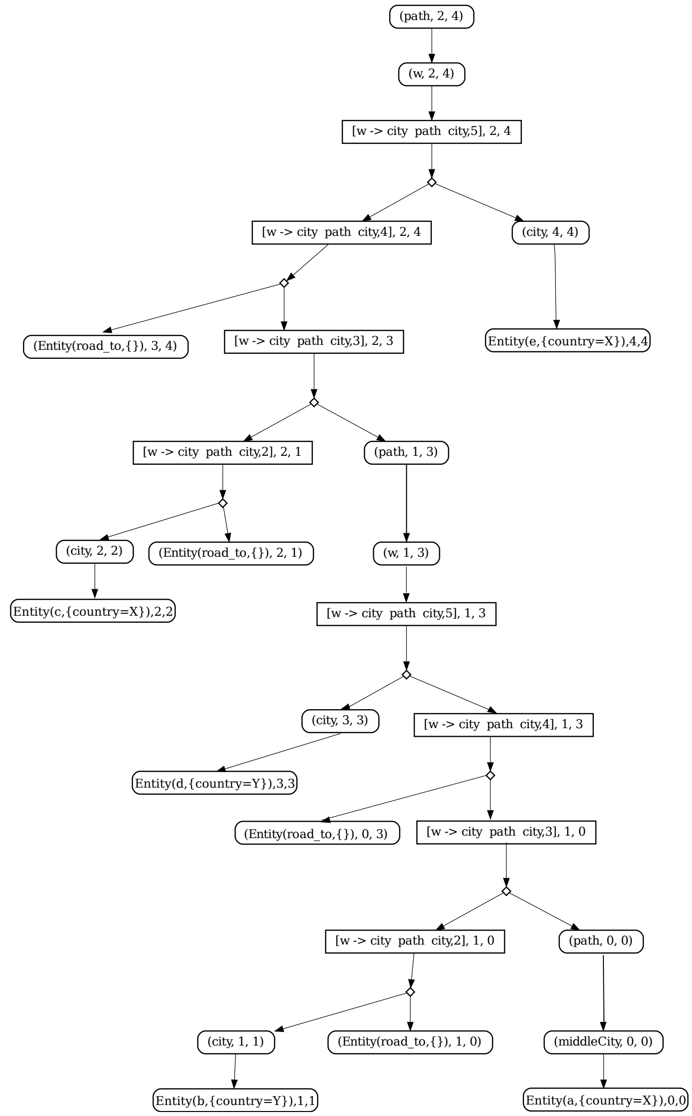


**Figure 4: SPPF: result of applying cities query to the graph 3**

```
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 6. Now when we execute that query we will get a

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

**Figure 5: Path query**

list which consists of all pairs of city's names which have a road between.

```
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 6: Path query**

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**

## 4.3 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 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 .7. 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 7: 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.

## 4.4 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.

## 5 EXAMPLES

In this section we introduce and describe some examples of our library usage. We show that combinators are expressive enough for realistic queries and allows to create generic queries easely.

### 5.1 Complicated Query to Map

Let's form a complex query for our city graph. Let us capture one city, let's say city with name $a$. Now having a city graph and captured vertex we would like to know all paths such if as $i$ city from begining of our path we visit country $X$ then as $i$ city from end of our path we visit country $X$ too. And also the middle city in our path is our captured city $a$. In a terms of combinators we can define our path as shown on fig. 8. Here `reduceChoice` is a function which transforms a list of queries to one query which is formed by reducing given list with `|` combinator. The `pathPart` query recursively defines a path of our way. Also, `middleCity` is a vertex query which parses our captured city $a$ and `roadTo` query parses a *roadTo* edge.

Now we would like, to get from our query only `city` combinator result. For that purpose let us modify it to make return result. In our library we have a `^` and `&` functions for that. Then we will have definition of our combinators as presented in fig. 10.

Now we execute our query. It is evident that for the graph presented on fig. 3 we can get only three paths which satisfies given criteria:

- single-vertex path $a$;
- $b \to a \to d$
- $c \to b \to a \to d \to e$

```
val countriesList = List("X", "Y")
val path =
  (reduceChoice(countriesList.map(pathPart)) |
    middleCity)
def pathPart(country: String) =
  syn(city(country) ~ roadTo ~ path ~
    roadTo ~ city(country))

val middleCity = V(_.value() == "a")
val roadTo = E(_.value() == "road_to")
def city(country: String) =
  V(_.country == country)
```

**Figure 8: Path query**

```
def reduceChoice(xs: List[Nonterminal]) =
  xs match {
    case x :: Nil     => x
    case x :: y :: xs =>
      syn(xs.foldLeft(x | y)(_ | _))
  }
```

**Figure 9: Reduce choice function implementation**

```
val middleCity =
  syn(V(_.value() == "a") ^^) & (List(_))
def pathPart(country: String) = syn(
  (city(country) ~ roadTo ~
    path ~ roadTo ~ city(country) & {
      case a ~ (b: List[_]) ~ Entity =>
        a +: b :+ c })
```

**Figure 10: Fixed queries**

A simplified SPPF for this query is presented in Fig. 4: rounded rectangles represent nonterminals and other rectangles represent productions. Every rectangle contains a nonterminal name or a production rule, as well as start and end nodes of the path in the input graph derived from the corresponding rectangle. Gray rectangles are start nonterminals.

### 5.2 Same Generation Query

Yet another example of first order functions usage is generalisation o.f classical same generation query which is one of basic context-free path queries. One of application of such queries is hierarchy analysing in RDF storages [35]. Let suppose that we have RDF graphs with two pairs of relation (each pair is relation and its revers): ($subClassOf$; $subClassOf^{-1}$) and ($type$; $type^{-1}$). We want to evaluate two queries which detect all pairs of nodes which are connected by path derivable in grammars $G_1$ (Fig. 11) and $G_2$ respectively (Fig. 12).

Of course, these queries can be written in Meerkat easely because it supports context-free queries: code is presented in Fig. 13 and Fig. 14.

$$\begin{array}{ll} 0: & S \longrightarrow subClassOf^{-1}\ S\ subClassOf \\ 1: & S \longrightarrow type^{-1}\ S\ type \\ 2: & S \longrightarrow subClassOf^{-1}\ subClassOf \\ 3: & S \longrightarrow type^{-1}\ type \end{array}$$

**Figure 11: Context-free grammar $G_1$ for query 1**

$$\begin{array}{ll} 0: & S \longrightarrow B\ subClassOf \\ 0: & S \longrightarrow subClassOf \\ 1: & B \longrightarrow subClassOf^{-1}\ B\ subClassOf \\ 2: & B \longrightarrow subClassOf^{-1}\ subClassOf \end{array}$$

**Figure 12: Context-free grammar $G_2$ for query 2**

```
val query1: Nonterminal = syn(
   "subclassof-1" ~ query1.? ~ "subclassof" |
   "type-1" ~ query1.? ~ "type")
```

**Figure 13: The same generation query (Query 1) in Meerkat**

```
val S = syn(
  "subclassof-1" ~ S ~ "subclassof")
val query2 = syn(S ~"subclassof")
```

**Figure 14: The same generation query (Query 2) in Meerkat**

As you can see, grammars and code representations for these two queries looks pretty similar. May we avoid code duplication and generalize them? Yes, we can and not only for these two queries. The function `sameGen` presented in Fig 15 is a generalization of the same generation query and is independent of the environment such as the input graph structure or other parsers and also uses a function `reduceChoice` presented in 9. It can be used for the creation of other queries, including the one presented in Fig 13: it is the result of the application of `sameGen` to the appropriate relations (which can be treated as opening and closing brackets). Another application of the `sameGen` is a Query 2, which can be founded in Fig. 17.

We show that parser combinators provide a simple and safe way to creation of generic queries. By using this ability, it may be possible to create a library of "standard templates" for most popular generic queries like same generation query or for domain specific queries (for example, for specific static code analysis problem).

## 5.3 Classical Movies Queries

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Mov!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MovMovMov! Mov! Mov! Mov! Mov! Mov! Mov! Mov! Mov! Mov! Mov! Mov! Mov!!!!!!!!!!!!!!!!!!!

```
def sameGen(brs) =
  reduceChoice(
    bs.map { case (lbr, rbr) =>
      lbr ~ syn(sameGen(bs).?) ~ rbr })
```

**Figure 15: Generic function for the same generations query**

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

**Figure 16: Query 1 as an application of `sameGen`**

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

**Figure 17: Query 2 as an application of `sameGen`**

# 6 EVALUATION

In this section, we present an evaluation of our graph querying library. We measure its performance on a classical set of ontology graphs [35]: 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 [6] and the Trails [12] 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.

## 6.1 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 [35]. 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 [6] for these graphs. The grammars for the queries are presented in Fig. 16 and Fig. 17.

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 [6] 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.

| 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**

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

**Figure 18: Context-Free grammar for the may-alias problem**

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

**Figure 19: Meerkat representation of may-alias problem grammar**

## 6.2 Static code analysis

Alias analysis is a fundamental static analysis problem [15]: it checks may-alias relations between code expressions and can be formulated as a context-free language (CFL) reachability problem [23] which is closely related to context-free path querying problem. In this analysis, a program is represented as a Program Expression Graph (PEG) [36]. 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 [36] is presented in Fig. 18. It contains two nonterminals **M** and **V** and allows us to make two kinds of queries for each of them.

- **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.

## 6.3 Comparison with Trails

Trails [12] 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.

# 7 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

| 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**

about their structure. This may simplify debugging and query result processing.

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 [19] reachability, but not context-free [34]. 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 [18] 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 [14], 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] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification inference using context-free language reachability. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 553–566.

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

[4] 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

[5] 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.

[6] 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

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

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

[9] 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.

[10] 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

[11] 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

[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] 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.

[14] 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

[15] 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

[16] 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

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

[18] Alexander Okhotin. 2001. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics* 6, 4 (2001), 519–535.

[19] 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

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

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

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

[23] 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

[24] 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

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

[26] 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.

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

[28] Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Science of Computer Programming* 78, 10 (2013), 1828–1844.

[29] Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. 2007. BRNGLR: a cubic Tomita-style GLR parsing algorithm. *Acta informatica* 44, 6 (2007), 427–461.

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

[31] 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

[32] 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

[33] 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.

[34] 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

[35] 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.

[36] 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