Parser Combinators for Context-Free Path Querying

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
s.v.grigoriev@spbu.ru

Abstract

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 using CFPQs. 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, Context-Free Language Reachability, Parser Combinators, Generalized LL, GLL, Neo4j, Scala

Scala '18, September 28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5836-1/18/09…$15.00
https://doi.org/10.1145/3241653.3241655

ACM Reference Format:
Ekaterina Verbitskaia, Ilya Kirillov, Ilya Nozkin, and Semyon Grigorev. 2018. Parser Combinators for Context-Free Path Querying. In *Proceedings of the 9th ACM SIGPLAN International Scala Symposium (Scala '18), September 28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3241653.3241655

1 Introduction

Graph querying is finding all paths in the graph which satisfy some constraints. If the constraints are specified with some language formalism, i.e. a grammar, it is called a language-constrained path query. The simplest query described with a grammar $S \longrightarrow a\ b$ being run against the graph in the Fig. 1 returns the only path 2, 3, 4 (shown in red).

A grammar $S \longrightarrow a\ S\ b \mid a\ b$ is a query for the paths of the form $a^n b^n$, where $n \geq 1$. Querying the graph in the Fig. 1 returns the infinite set of paths one 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.

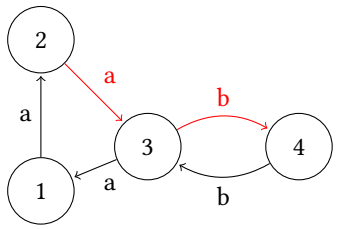

Figure 1. An example of input graph

Most existing graph traversing/querying languages, including SPARQL [Prud et al. 2006], Cypher [1], and Gremlin [Rodriguez 2015] support only regular languages as constrains. 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 [Sevon and Eronen 2008], static code analysis [Bastani et al. 2015; Pratikakis et al. 2006; Reps 1997; Yan et al. 2011;

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

Zheng and Rugina 2008], and RDF processing [Zhang et al. 2016]. Although there is a lot of problem-specific solutions and theoretical research on CFPQs [Barceló et al. 2013; Grigorev and Ragozina 2017; Hellings 2014, 2015; Mendelzon and Wood 1995; Reutter et al. 2015; Sevon and Eronen 2008; Yannakakis 1990], cfSPARQL [Zhang et al. 2016] 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 transparent and native access to data sources. One way to achieve this 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, 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 [Bultan et al. 2018]. Such techniques as the Object Relationship Mapping (ORM) or Language Integrated Query (LINQ) [Cheney et al. 2013; Kumamoto et al. 2015; Meijer et al. 2006] partially 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 they are called in the area of static code analysis. In [Reps 1997; Reps et al. 1995] Thomas Reps shows that the wide range of static code analysis problems can be formulated in terms of CFL-reachability in the graph. This framework is used for such problems as the taint analysis [Huang et al. 2015], the alias analysis [Wang et al. 2017; Yan et al. 2011; Zheng and Rugina 2008], the label flow analysis [Pratikakis et al. 2006], and the fix locations problem [Dan et al. 2017]. What we propose in the paper can be viewed as a core of such framework since it provides both problem and domain independent mechanism for CFPQ evaluation.

We view parser combinators as the best way to integrate context-free language specifications into a general-purpose programming language. Parser combinators provide not only 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 [Kröni and Schweizer 2013]. 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 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 [Izmaylova et al. 2016], the authors demonstrate a set of parser combinators which handles arbitrary context-free grammars by using ideas of the Generalized LL [Scott and Johnstone 2010] algorithm (GLL). Meerkat [2] parser combinators library implements the ideas from the paper [Izmaylova et al. 2016] and provides the parsing result in a compact form as a Shared Packed Parse Forest [Rekers 1992] (SPPF). SPPF is a suitable finite structural representation of a CFPQ result, even when the set of paths is infinite [Grigorev and Ragozina 2017]. All the paths can be extracted from the SPPF—in the form of the corresponding derivation trees—and further analysis can be done. It is also possible to run some further processing over the SPPF itself—not extracting the paths explicitly.

In this paper, we compose these ideas and present a set of parser combinators for context-free path querying which handles arbitrary context-free grammars and provides 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 works on both arbitrary context-free grammars and arbitrary graphs and provides a finite structural representation of the query result.
2. We implement the parser combinators library in Scala. This library provides 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.

This paper is organized as follows. We introduce a formal definition of the CFPQ problem in section 2, and we provide a basic description of the Meerkat library and SPPF data structure in section 3. We describe our solution in section 4. In section 5 we present and discuss a set of classical queries (the same generation query, the queries to a movie dataset [4]) written with our library. Evaluation of the library is described in section 6. Finally, In section 7 we conclude and discuss possible directions for further research.

[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

[4]The movie database is a traditional dataset for graph databases. Detailed description is available here: https://neo4j.com/developer/movie-database/. Access date: 16.01.2018

2 Context-Free Path Querying Problem

In this section, we formally describe the context-free path querying problem and the context-free reachability problem.

First, we introduce the necessary definitions.

- A context-free grammar is a quadruple $G = (N, \Sigma, P, S)$, where N is a set of nonterminal symbols, Σ 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: $\mathcal{L}(G) = \{\omega \mid S \Rightarrow^*_G \omega\}$.
- A 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 $E \subseteq V \times L \times V$. Note that there are not parallel edges with equal labels: if $(v, l_1, u) \in E$ and $(v, l_2, u) \in E$, then $l_1 \neq l_2$.
- $tag : E \to L$ is a function which returns a tag of an edge.
$$tag((\_, l, \_)) = l$$
- $\oplus : L^+ \times L^+ \to L^+$ denotes a tag concatenation operation.
- Ω 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 from the input graph M which are also strings derivable in the grammar G:

$$\{p \mid p \text{ is a path in } M, \Omega(p) \in \mathcal{L}(G)\}.$$

The CFL reachability problem is pretty similar, but here one is only concerned with the existence of the paths derived by the grammar. It is formulated as follows:

$$\{(v_0, v_n) \mid p \text{ is a path in } M, \Omega(p) \in \mathcal{L}(G),$$
$$p = v_0 \to \cdots \to v_n\}.$$

Note that the query result 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 the query result 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 [Kröni and Schweizer 2013], but it still suffers from the common issue with left-recursive definitions. A general parser combinators library Meerkat [Izmaylova et al. 2016], implemented in the Scala programming language, removes this restriction by using memoization, continuation-passing style, and the ideas of Johnson [Johnson 1995]. Meerkat converts parsers into a memoized versions of themselves. The memoization routine stores parsing results the first time they are computed and reuses them every time they are needed again. Every time a new parsing result for some combinator is calculated, all the combinators, which are dependent on it, are recomputed. This way Meerkat employs left-recursive parser combinators, and it is also crucial for the handling of the cycles in the input graphs.

Meerkat supports the arbitrary (left-recursive and ambiguous) context-free specifications; it also supports the specification of action codes and provides 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 [Grigorev and Ragozina 2017]. The worst case time and space complexity of the solution are cubic.

A Meerkat specification of the language $\{a^n b^n \mid n \geq 1\}$ is `val S = syn("a" ~ S.? ~ "b")`. Here `syn` is a macro which creates a parser: for example, it transforms string literals into parsers for those strings. The tilde `~` stands for a sequential parser combinator, and the question mark `?` describes optional parsing. Other combinators available in the Meerkat library are shown in Table 1.

It is shown in [Grigorev and Ragozina 2017] that the Generalized LL parsing algorithm [Scott and Johnstone 2010] can be generalized to process CFPQs effectively and the query result can be finitely represented. As the Meerkat library is closely related to the Generalized LL algorithm, 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 are described in the next section.

3.1 SPPF

Parsing of a string with respect to an ambiguous grammar can result in several derivation trees for a single string. The set of derivation trees is named a *derivation forest*. To store a derivation forest efficiently, the generalized parsing algorithms utilize a *Shared Packed Parse Forest* proposed by Joan Rekers [Rekers 1992]. The most efficient compact representation of derivation forests is a Binarized Shared Packed Parse Forest (we will abbreviate it to SPPF) [Scott et al. 2007]. The GLL algorithm, which employs this structure, achieves the worst-case cubic space complexity [Scott and Johnstone 2013].

Binarized SPPF is a directed graph, each node of which has one of the four types described below. Almost every node of the SPPF is decorated with the *extension*: a pair (i, j) where i is a start position of a substring derivable from the node and j—its end position.

- A **terminal node** is labelled (T, i, j).
- A **nonterminal node** is labelled (N, i, j). This node denotes that there is at least one derivation $N \Rightarrow^*_G \omega[i \ldots j-1]$—a substring of the input from the i-th to

the j-th position. Every derivation tree for the given substring and nonterminal can be extracted by left-to-right top-down traversal of SPPF started from the respective node.
- An **intermediate node**: a special kind of node used for the binarization of the SPPF. These nodes are labelled with (t, i, j), where t is a grammar slot.
- A **packed node** is labelled $(N \rightarrow \alpha, k)$, where k is a position in the input of the right end of the leftmost subtree of this node. A subgraph with the root in such node is, in turn, a parse forest for which the first production is $N \rightarrow \alpha$.

An example of SPPF is presented in Fig. 3. We removed redundant intermediate and packed nodes for simplicity and to decrease the size of the figure.

SPPF finitely represents a possibly infinite set of paths in the context of the language-constrained graph querying [Grigorev and Ragozina 2017]. Since the SPPF stores derivation trees for all paths, it is useful for postprocessing and further understanding of the query results. In static code analysis, for example, it is possible to map paths back onto the source code thus providing a human-readable result.

4 Parser Combinators for Path Querying

Parser combinators are a way to specify both a language syntax and a parser for it in terms of higher-order functions. A parser in this framework is a function which consumes a prefix of input and returns either a parsing result or an error if the input is erroneous. Parser combinators compose parsers to form more complex parsers. A parser combinators library usually provides a set of basic parser combinators, such as a combinator of the sequential application or the choice combinator, but there can also be user-defined combinators. Most parser combinators libraries, including the Meerkat library, can only process the linear input—strings or streams. We modify the Meerkat library to work on the graph input. The following ideas are at the core of the modification.

The intersection of a context-free and a regular language is context-free. There are several constructive proofs of this fact. The proposed solution is yet another constructive proof with the SPPF as a user-friendly representation of the context-free grammar for the intersection.

Linear input can be regarded as a linear directed graph with symbols of the input labelling the edges. A conventional parser moves a pointer in the input from the position i to the position $i + 1$ and creates a new state when a token between the i-th and the $i + 1$-th positions matches what is required in the grammar. In case of graph processing, there are possibly multiple ways to move from the current vertex i, and it is possible to produce multiple new states. Generalized parsing is designed to handle the production of multiple new states optimally, thus it is suitable to handle graph processing.

Matching a token in the input can be viewed as a predicate, for example, $p_c(x) = x == c$. We can generalize this observation allowing matching of an edge label of an arbitrary type with a predicate of some sort. If vertices of the graph contain any data of interest, we can treat them in the similar fashion as the edges. The formal definition of the context-free language constrained path querying changes only in a way a string is collected along a path in the input graph: the data stored in the vertices should also be taken into consideration.

Handling cycles in the input graphs imposes two challenges: not to get stuck in an infinite loop while processing the positions and if a new parsing state appears at some position, all paths which pass through this state, should be accounted for. The Meerkat memoization routine solves both of these challenges. Parsing from a parser state at each position is only run if it has never been run before, so since there is only a finite number of parsing states and positions in the input, parsing terminates. Appearing of the new parsing state at a position which has been processed before triggers processing of every path possibly affected by it. Thanks to the memoization, each derivation of each subpath is analyzed only once, so there is no significant overhead at re-running.

Querying process in our library is inherited from generalized parsing and is done in two steps. The first step is "parsing": the construction of the SPPF which contains all derivation trees for the paths satisfying syntactic constraints. The second step is the application of semantic actions which retrieves the necessary additional data about the paths from the SPPF

4.1 The Set of Combinators

In this section, we demonstrate the set of combinators by example. The input graph, which represents a map, is presented in Fig. 2. In the input graph, there are some cities connected by one-way roads depicted by the edges labelled *road_to*. Each city is labelled by its name and a country it belongs to.

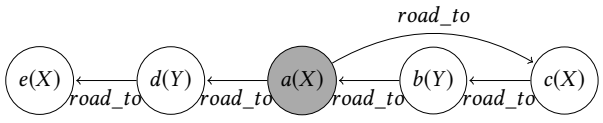

Figure 2. Example graph. Vertex labels are in the form "city-name (country-name)"

Two basic building blocks of queries are the combinators for dealing with edges and vertices.

- `V[N](predicate: N => Boolean)` the combinator for processing vertices, where `N` is a type of the node label. Parsing with this combinator succeeds iff the vertex satisfies the predicate.

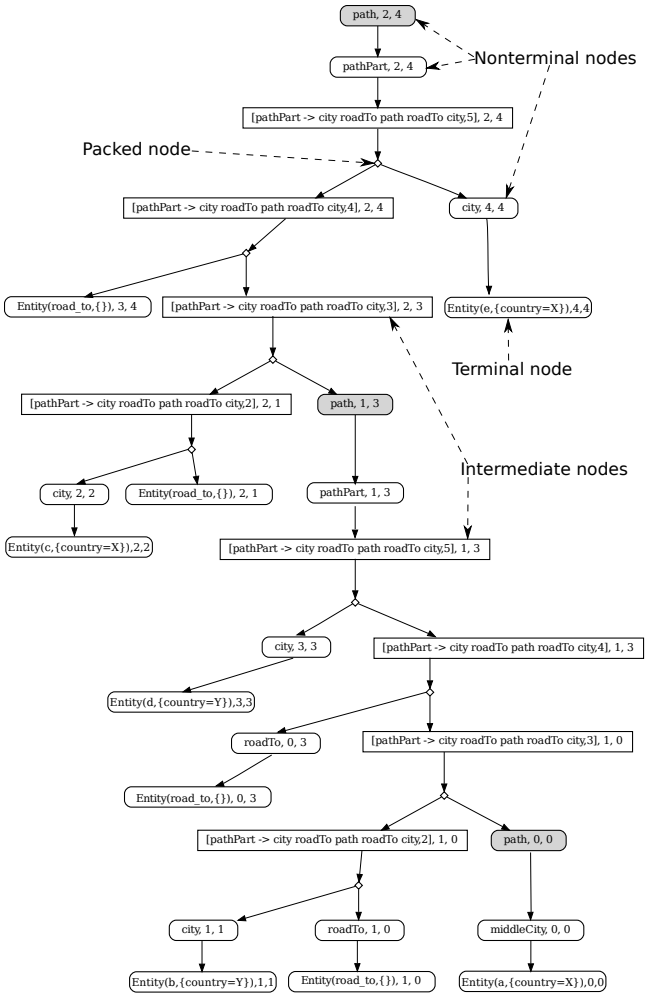

Figure 3. SPPF: result of applying cities query to the graph 2

- `E[L](predicate: L => Boolean)` the combinator for processing edges, where `L` is a type of the edge label. Parsing with this combinator succeeds iff the edge satisfies the predicate.

To select cities which belong to some country, we can use the function `V`: `V[N]((e: Entity) => e.country = "County Name")`. Here `Entity` is a property container for both edges and vertices. By using the `Dynamic` trait, all accesses to properties (like `(e: Entity).country`) are converted to the accesses to the properties of either a vertex or an edge. For the sake of simplicity, we will omit `Entity` type specifications for predicates. To query the graph for the paths from a city in the country X to a city in the country Y, we need to sequentially compose the combinators for selecting the appropriate cities. A sequential combinator `~` does just that: it sequentially applies two queries one after the other. Let us denote a query for retrieving a city from the specific country `city(name: String)` and a query for retrieving road edges `roadTo`. With this denotation, a query `city("X") ~ roadTo ~ city("Y")` returns the requested set of paths from the graph. The query with all necessary subqueries is the following[5].

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

The next step after writing the query is to implement a function to retrieve the actual data about the paths. If we care only about the names of the cities, we can return a pair of the cities for each path. First, we modify the query for vertices by adding the semantic action to it using the combinator `^`:

```
def city(name: String) =
  syn(V(e.value() == name) ^ (_.value))}
```

Then we need to map a path to a pair of cities: this is done with the combinator `&`. The complete query returns the sequence of pairs of cities which have the road between them:

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

The whole set of basic combinators, which our library provides, is presented in Table 1. There are two kinds of combinators: the first kind combines parsers to form new parsers; meanwhile the second one is dedicated to the processing of the query result. Whenever a string is used within a query, a parser which matches that string is generated implicitly.

Table 1. Basic combinators

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

4.2 Generic Interface for Input

The combinators in our library are independent of the input representation. It is enough to specify two basic combinators which handle vertices and edges. Vertex handling is checking whether the vertex satisfies the given predicate. In case of the edges, one needs to check which of the edges outgoing

[5]Underscore in anonymous lambda functions serves as a placeholder for an argument: `(_.country == country)` is equavalent to `(x => x.country == country)`

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

Figure 4. Generalized input interface

from the given vertex satisfy the given predicate. These two functions form the trait for the input (Fig. 4). It has two type parameters: the type of edge labels `L` and the type of vertices labels `N`. Since the required functions are simple, we believe it is possible to support most storages of graph-structured data. We supported several different input sources:

- Neo4jInput: input source for the graph database Neo4j;
- GraphxInput: input source for the graph presented in memory using the GraphX library;
- LinearInput: input source for the linear input data such as the ordinary strings.

4.3 Semantic Actions

Every path, which the query produces, has a derivation tree stored in the SPPF. The derivation tree is a very rich data structure which can be hard to understand. The library provides a mechanism of semantic actions to retrieve the data useful for the user. It is a way to apply some function to a parsed token or a subsequence.

Semantic action binder for the tokens—vertices and edges alike—is `^`. The most common use for it is to extract properties from the token and combine them in some fashion.

```
// 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] {...
```

For the combination of parsers, there is a `&` binder. Being applied to a sequence of tokens, it can collect and process the data returned by the terminal parsers.

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

They both produce a new parser that parses the input exactly like the given parser but also have a bound function. The function is referenced in each SPPF node created by the corresponding parser.

The way semantic actions are executed has mostly remained the same as in the original Meerkat library: semantic actions are first executed for the children of the current node, then the results are collected and passed to a semantic action of the current node. If there are ambiguous nodes in the SPPF, the original Meerkat library just throws an exception. In our case, ambiguity can arise not only when there are multiple derivations of a string, but ambiguous nodes can also represent several different subpaths which are derived from the corresponding nonterminal. We chose to provide a way to extract the derivation trees from the SPPF lazily since the number of the paths can be infinite. Unambiguous trees are yielded with a breadth-first search.

The composition of the extraction of trees and the semantic action execution is called `executeQuery`. It parses the input graph from all positions, produces a list of SPPF roots, extracts all derivations from every root, executes semantic actions and returns a lazy stream of results.

5 Examples

In this section, we describe some examples of queries written with the library proposed. We show that the combinators are expressive enough for realistic queries and also ease their creation.

5.1 Complicated Query to Map

We would like to search for all the routes which pass through a fixed city *a* such that the countries of the cities on the route form a palindrome. This means that if a route starts at the city of the country *X*, then the last visited city should also belong to the country *X*. We also demand that the fixed city *a* is in the middle of the route. This query can be written as follows. A subquery `middleCity` matches the fixed city *a*, and a query `roadTo` matches a *roadTo* edge.

```
val countries = List("X", "Y")
val path = reduceChoice(countries.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)
```

A combinator `reduceChoice` reduces a list of queries to a single query by means of the combinator of choice`|`. The implementation of the `reduceChoice` is straightforward.

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

To filter out all the data but the list of the cities on the route, we can add the semantic actions.

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

Executing the query for the graph in Fig. 2 returns the only three routes, which satisfy our restrictions.

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

A simplified SPPF for this query is presented in Fig. 3. Rounded rectangles represent nonterminals, and other rectangles represent productions. Every rectangle vertex contains a nonterminal name or a production rule, as well as the start and the end nodes in the input graph for the path derived from the corresponding SPPF vertex. The start nonterminals are drawn in grey.

5.2 Same Generation Query

The generalization of the classical same generation query benefits from utilizing the first-order functions for querying. Such a query can be used for the hierarchy analysis in RDF storages [Zhang et al. 2016]. Let's consider RDF graphs which have two pairs of relations between objects: ($subClassOf$; $subClassOf^{-1}$) and ($type$; $type^{-1}$). Each relation has its reverse denoted by the -1 superscript. To search for the vertices which are on the same level of hierarchy one can use the grammars in Fig. 5 and Fig. 6.

$$\begin{array}{rcl} S & \to & subClassOf^{-1}\ S?\ subClassOf \\ & | & type^{-1}\ S?\ type \end{array}$$

Figure 5. Context-free grammar for query 1

$$\begin{array}{rcl} S & \to & B?\ subClassOf \\ B & \to & subClassOf^{-1}\ B?\ subClassOf \end{array}$$

Figure 6. Context-free grammar for query 2

These two queries are context-free, so they can be easily written in Meerkat.

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

val B = syn("subclassof-1" ~ B.? ~ "subclassof")
val query2 = syn(B.? ~ "subclassof")
```

The implementations of the queries are similar, and we can get rid of the code duplication by generalizing them. Both the first query and the nonterminal B in the second query denote the languages of nested brackets: the first one has two types of brackets, and the second one—only one. We can implement a parser combinator for the nested brackets language, which is parameterized by the bracket pairs. The combinator `sameGen` is exactly that. It generalizes the same generation queries and is independent of the environment such as the input graph structure or other parsers. Note that `sameGen` does not limit "brackets" to be just string literals: one can use arbitrary parsers.

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

Fig. 7 shows how the same generation queries can be implemented as the application of the `sameGen` combinator to the appropriate relations.

```
val query1 = syn(sameGen(List(
    ("subclassof-1", "subclassof"),
    ("type-1", "type"))))
val query2 = syn(
  sameGen(List(("subclassof-1", "subclassof"))) ~
   "subclassof")
```

Figure 7. Queries 1 and 2 as the applications of `sameGen`

We illustrated that the generic queries are easily written by means of the parser combinators. It is possible to create a library of standard templates for most popular generic queries, such as the same generation query or other domain-specific queries (for example, for specific static code analysis problem).

5.3 Classical Movies Queries

We also implemented some queries to the movie database used in the Neo4j tutorial [6]. The database contains data about movies, actors, directors and relations between them. In this set of queries, we demonstrate more semantic actions for the processing of results.

Several helper functions were implemented to simplify the processing of the nodes and the edges.

```
def LV(labels: String*) =
  V(e => labels.forall(e.hasLabel))
def outLE(label:String) = outE(_.label() == label)
def inLE (label:String) = inE (_.label() == label)
```

A vertex in the Neo4j database can have several labels while an edge can only have one; thus the functions for the handling of vertices and edges accept a different number of arguments. The function `LV` matches a vertex if it has each of the labels passed as the argument. The functions `outLE` and `inLE` match an edge if its label is the same as the argument. The difference between these two functions is in what direction the edge should go to be accepted. If the direction of the edge is supposed to coincide with the direction of the path exploration, one should use `outLE`, otherwise—`inLE`.

Having the helper functions implemented, we can start building queries. The first query selects *actors who played*

[6] The set of classical queries to movie dataset in Cypher language: https://neo4j.com/developer/movie-database/.

in some film, in our example in "Forrest Gump". For every actor, the query selects the name and the birthplace. Compare the Cypher (Fig. 8) and the Meerkat versions (Fig. 9) of the query. The structure of them is similar: the first part specifies the path and the second part—the values to return; in the Meerkat version we use semantic actions to calculate it.

```
MATCH (m:Movie {title: 'Forrest Gump'})
      <-[:ACTS_IN]-(a:Actor)
RETURN a.name, a.birthplace;
```

Figure 8. *Actors who played in some film* query in Cypher

```
val query =
  syn((
    (LV("Movie")::V(_.title == "Forrest Gump")) ~
    inLE("ACTS_IN") ~
    syn(
      LV("Actor") ^ (e =>
        (e.name, e.birthplace)))) &&)
executeQuery(query, input)
```

Figure 9. *Actors who played in some film* query in Meerkat

The *most prolific actors* query (Fig. 10) returns the ordered list of ten actors, who starred in the largest number of movies. Here we need to use the postprocessing of the paths set to express ordering: `executeQuery` returns a lazy set of paths which can be processed by using standard Scala functions as shown in Fig. 11.

```
MATCH (a:Actor)-[:ACTS_IN]->(m:Movie)
RETURN a, count(*)
ORDER BY count(*) DESC LIMIT 10
```

Figure 10. *Most prolific actors* query in Cypher

```
val query =
  syn((
    syn(LV("Actor") ^^) ~ outLE("ACTS_IN") ~
    LV("Movie")) & (a => (a.name, a.toInt)))
executeQuery(query, input)
  .groupBy {case (a, i) => i}
  .toIndexedSeq
  .map {case (i, ms) => (ms.head._1, ms.length)}
  .sortBy {case (a, mc) => -mc}}
  .take(10)
```

Figure 11. *Most prolific actors* query in Meerkat

The query presented in Fig. 12 searches for the movies which friends of the fixed user rated above 3 stars. Then it composes the recommendation which consists of the title of the movie, the rate, the name of the friend and the comment they left. It is necessary to use subqueries to write this query in Meerkat. First of all, we specify the `user` subquery to find the user with the specified login. Then we define the `friendsWith` subquery. Finally, we combine these subqueries to form the resulting query (Fig. 13).

```
MATCH (u:User {login: 'adilfulara'})-
      [:FRIEND]-(f:Person)-[r:RATED]->(m:Movie)
WHERE r.stars > 3
RETURN f.name, m.title, r.stars, r.comment;
```

Figure 12. *Mutual friend recommendations* query in Cypher

```
val user = syn(
  LV("User")::V(_.login == "adilfulara"))
val friendsWith =
  syn(inLE("FRIEND") | outLE("FRIEND"))
val query = syn((
  user ~ friendsWith ~ syn(LV("Person") ^^) ~
  syn(outLE("RATED") ^^) ~ syn(LV("Movie") ^^)) &
    {case p ~ r ~ m =>
       (p.name,
        m.title,
        r.stars.toInt,
        if (r.hasProperty("comment")) r.comment
        else "")})
executeQuery(query, input)
  .filter {case (_, _, s, _) => s > 3}
```

Figure 13. *Mutual friend recommendations* query in Meerkat

We showed that our library is expressive enough to formulate realistic queries. The main drawback of our library as compared to the Cypher language is that all additional logic such as filtering, sorting or grouping has to be implemented manually as a separate step.

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 [Zhang et al. 2016]: 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 [Grigorev and Ragozina 2017] and the Trails [Kröni and Schweizer 2013] library for graph traversals. We also show how may-alias static code analysis can be done by means of the 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.

Table 2. Comparison of Meerkat, Trails and GLL performance on ontologies

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

6.1 Ontology Querying

Querying for ontologies is a well-known graph querying problem. We evaluate our library on some popular ontologies in the form of RDF files from the paper [Zhang et al. 2016]. 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 [Grigorev and Ragozina 2017] for these graphs. The grammars for the queries are presented in Fig. 7.

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

The Meerkat-based and the GLL-based [Grigorev and Ragozina 2017] solutions show the same results (column *#results*) and the Queries 1 and 2 run at least 1.5 times faster on the Meerkat-based solution than on the GLL-based. Querying the Neo4j database is as performant as the querying the graphs located in the RAM.

6.2 Static Code Analysis

Alias analysis is a fundamental static analysis problem [Marlowe et al. 1993]: it checks may-alias relations between code expressions and can be formulated as a context-free language (CFL) reachability problem [Reps 1997], which is closely related to the context-free path querying problem. In this analysis, a program is represented as a Program Expression Graph (PEG) [Zheng and Rugina 2008]. Vertices in a PEG correspond to program expressions and edges are relations between them. There are two types of edges possible while analyzing source code written in the C programming language: **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 the may-alias problem from [Zheng and Rugina 2008] and its implementation in Meerkat are shown in Fig. 14. There are two nonterminals **M** and **V**, which we can query for. **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.

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

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

Figure 14. Context-Free grammar and its Meerkat representation for the may-alias problem

We run the **M** and **V** queries on some open-source C projects. We used Crystal[7] to construct Program Expression Graph and ran the queries over it. The results are shown in Table 3. We conclude that our solution is expressive and performant enough to be used for static analysis problems, which can be expressed as CFPQs.

[7]Crystal: a program analysis system for C https://www.cs.cornell.edu/projects/crystal/. Access date: 30.07.2018.

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

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 4. Running regular queries using Meerkat and Cypher

Query	Neo4j + Meerkat (ms)	Neo4j + Cypher (ms)
Actors who played in some film	89.78	11.31
Most prolific actors	2333.37	189.32
Mutual friend recommendations	106.86	17.26
Directed more than 2 films, acted in more than 10	348.11	44.82

6.3 Classical Movies Queries

To examine the applicability of our library for the real data processing, we evaluate queries presented in section 5 on the classical movie database which contains more than 60000 nodes and more than 100000 edges.

All queries are performed using the Neo4j database connected to Meerkat. During the evaluation of these queries, caching in Neo4j was disabled to simplify time measurement. The results are presented in Table 4.

Our library implements a general parsing algorithm which supports arbitrary context-free queries. Unfortunately, when the queries are regular, there is a significant overhead for processing them. The queries mentioned in this section are all regular, thus it is not a surprise that our implementation is dramatically slower than the native Neo4j solution. Besides the overhead itself, our library does not take any advantages from the internal Neo4j optimizations.

Despite the seeming failure of this test, we still can conclude that the combinators can process quite big datasets in a reasonable time: 6 times bigger in terms of the number of vertices than the other tests. It also shows that further optimization of the implementation is justified.

6.4 Comparison with Trails

Trails [Kröni and Schweizer 2013] 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 the basic traversers. Both Trails and Meerkat-based solution support parsing of the graphs located in RAM, so we compare the performance of the Trails and the 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 the Meerkat-based solution.

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

7 Conclusion

We propose a native way to integrate a language for context-free path querying into a general-purpose programming language. Our solution handles arbitrary context-free grammars and arbitrary input graphs. The proposed approach is language-independent and may be implemented in nearly every general-purpose programming language. We implement it in Scala as a modification to the parser combinator library Meerkat and show that our approach can be applied to the real world problems.

We can propose some possible directions for the future work. To improve the performance and investigate the scalability of the solution, we plan to implement a parallel single machine and distributed GLL. It is a challenge from both the theoretical and the practical standpoint.

Some important problems in the realm of the static code analysis cannot be expressed in terms of context-free path querying. For example, the context-sensitive data-dependence analysis may be precisely expressed in terms of the linear-conjunctive language [Okhotin 2003] reachability but not context-free [Zhang and Su 2017]. How to support the arbitrary conjunctive grammars is also worth research. This technique can be employed as a static analysis framework.

Acknowledgments

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

References

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.

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

Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. 2018. String Analysis for Software Verification and Security.

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

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.

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

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

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

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.

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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.

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

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.

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