# Parser-Combinators for Context-Free Path Querying

Sophia Smolina
Electrotechnical University
St. Petersburg, Russia
sofysmol@gmail.com

Ekaterina Verbitskaia
Saint Petersburg State University
St. Petersburg, Russia
kajigor@gmail.com

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

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

## ABSTRACT

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

## CCS CONCEPTS

• **Information systems** → **Graph-based database models**; **Query languages for non-relational engines**; • **Software and its engineering** → *Functional languages*; • **Theory of computation** → *Grammars and context-free languages*;

## KEYWORDS

Graph Databases, Language-Constrained Path Problem, Context-Free Path Querying, Parser Combinators, Generalized LL, GLL, Neo4J, Scala

**ACM Reference Format:**
Sophia Smolina, Ekaterina Verbitskaia, Ilya Kirillov, and Semyon Grigorev. 2018. Parser-Combinators for Context-Free Path Querying. In *Proceedings of Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) 2018 (GRADES-NDA'18).* ACM, New York, NY, USA, Article 4, ?? pages. https://doi.org/10.475/123_4


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

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

## 1 INTRODUCTION

One useful type of graph queries is language-constrained path queries [**?** ]. There are several languages for graph traversing/-querying which support constraints formulated in terms of regular languages. For example SPARQL [**?** ], Cypher [1], and Gremlin [**?** ]. In this work, we are focused on context-free path queries (CFPQs) which use context-free languages for constraints specification and are used in bioinformatics [**?** ], static code analysis [**? ? ? ? ?** ], and RDF processing [**?** ]. There are a lot of problem-specific solutions and theoretical research on CFPQs [**? ? ? ? ? ? ? ? ?** ]. Among them, cfSPARQL [**?** ] is a single known graph query language to support CF constraints. Generic solution for the integration of CFPQs into general-purpose languages is not discussed enough.

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

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

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

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

the language class supported or restrictions are not discussed. This point is very important, because conventional combinators implement top-down parsing and cannot handle left-recursive and ambiguous grammars.

In [**?** ], authors demonstrate a set of parser combinators which can handle arbitrary context-free grammars by using ideas of Generalized LL [**?** ] (GLL). Meerkat [2] parser combinators library is based on [**?** ] and provides result of parsing in a compact form as Shared Packed Parse Forest [**?** ] (SPPF). It is showed that SPPF is a suitable finite structural representation of a CFPQ query result, even if the set of paths is infinite [**?** ], which can be used for paths extraction, queries debugging and processing of result.

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

(1) We show that it is possible to create parser combinators for context-free path querying which work on both arbitrary context-free grammars and arbitrary graphs and provide a finite structural representation of the query result.
(2) We provide the implementation of the parser combinators library in Scala. This library provides an integration to Neo4j [3] graph database. The source code is available on GitHub:https://github.com/YaccConstructor/Meerkat.
(3) We perform an evaluation on realistic data. Also, we compare the performance of our library with another GLL-based CFPQ tool and with the Trails library. We conclude that our solution is expressive and performant enough to be applied to the real-world problems.

## 2 GENERALIZED LL

Sott, Ali, Meerkat

handle arbitrary context-free grammar, cubic time complexity.

GLL-based combinators.

Meerkat library is a general parser combinators library; by using memoization, continuation-passing style and the ideas of Johnson [**?** ], it supports arbitrary context-free specifications.

### 2.1 SPPF

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

Binarized Shared Packed Parse Forest (SPPF) [**?** ] 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 [**?** ].

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 $(N \rightarrow \alpha, k)$. Subgraph with "root" in such node is one possible derivation from nonterminal $N$ in case when the parent is a nonterminal node labeled with $(\diamond (i, N, j))$.

An example of SPPF is presented in figure **??**. We remove redundant intermediate and packed nodes from the SPPF to simplify it and to decrease the size of the structure.

### 2.2 GLL for CFPQ

Our work [**?** ]. It is possible to use GLL for CFPQ. String-embedded querying is a ugly solution.

As the Meerkat library is closely related to the Generalized LL algorithm and since GLL can be generalized for context-free path querying [**?** ], it is also possible to adapt Meerkat library for graph querying. It can be done by providing a function for retrieving the symbols which follow the specified position and utilizing it in the basic set of combinators. Detail described below.

## 3 PARSER COMBINATORS FOR PATH QUERYING

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

### 3.1 Generic interface

General input interface, as pointed in Trails

### 3.2 The set of combinators

The basic combinators our library provides are presented in table 1. Parsers for matching strings are implicitly generated whenever a string is used within a query. The classical same generation query [**?** ] can be written using the library as presented in Fig. 1.

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

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

[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

| 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**

The most exciting feature of our library is that queries can be used as first-class values which means greater generalization and composition. The function `sameGen` presented in Fig 3 is a generalization of the same generation query and is independent of the environment such as the input graph structure or other parsers. It can be used for the creation of other queries, including the one presented in Fig 1: 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. 6.

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

**Figure 2: Query 1 as an application of `sameGen`**

```
val g = new AnyRef {
      val P = syn(E((_: Int) > 0))
      val N = syn(E((_: Int) < 0))
      val S: Nonterminal[Int] = syn(P ~ N ~ S | P ~ N)
    }

 override def createParser: AbstractCPSParsers.AbstractSymbol[String, _, _] = {
    val num = V((_: String) forall Character.isDigit)
    syn(num ~ "+" ~ num)
  }
```

**Figure 3: Example of predicates usage**

## 3.3 An Example

Let's assume we have a graph which consists of vertices of two types: movie vertex labeled with a movie's title and actor vertex labeled with actor's name. An actor can stars in a movie. This relation shown in graph as an edge with label *stars_in*. Such graph is shown on 4. Let us capture one movie and one actor: Indiana Jones, and Harrison Ford, respectively. Now having that graph and two fixed vertices we would like to know all actors such they stars

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

**Figure 4: Generic function for the same generations query**

in Indiana Jones movie and also have at least another one common movie with Harrison Ford. In a terms of combinators we can define such query as the following:

```
val actors  = syn(fixedActor ~ starsIn ~ fixedMovie ~
   hasActor ~ actor ~ starsIn ~
   movie ~ hasActor ~ fixedActor)
```

**Figure 5: Actor combinator**

Here `actor` and `movie` is a vertex combinator which parses a vertex which is a actor or movie which is not captured one, respectively. And `fixedActor`, `fixedMovie` combinators stands for captured one. Also,`starsIn` and `hasActor` are edge combinators which parses edges with corresponding label on them. Those combinator can be defined the following way:

```
val actor = syn(V(e => e.ntype == "actor" &&
  e.value() != "Harrison␣Ford"))
val movie = syn(V(e => e.ntype == "movie" &&
  e.value() != "Indiana␣Jones"))
val starsIn = syn(E(e => e.value() == "stars_in"))
val hasActor = syn(E(e => e.value() == "has_actor"))
val fixedActor = syn(V(e => e.ntype == "actor" &&
  e.value() == "Harrison␣Ford"))
val fixedMovie = syn(V(e => e.ntype == "movie" &&
  e.value() == "Indiana␣Jones"))
```

**Figure 6: Combinators implementation**

Now we would like, to get from our query only `actor` combinator result. For that purpose let us modify it to make return result. In out library we have a `^^` and `&&` functions for that. Then we will have the following definition of our combinators:

```
val actor = syn(V(e => ...)^^)
val actors = syn((fixedActor ~ ... ~ fixedActor)&&)
```

**Figure 7: Fixed combinators**

Now we execute our query. For the graph presented on fig. 4 we will get the only actor which satisfies given criteria Julian Glover.

A simplified SPPF for this query is presented in Fig. 5: 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.

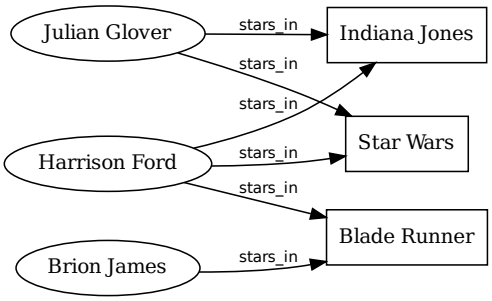


**Figure 8: Example Input graph**

## 4 EVALUATION

In this section, we present an evaluation of our graph querying library. We measure its performance on a classical set of ontology graphs [**?** ]: 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 [**?** ] and the Trails [**?** ] library for graph traversals. We also show how may-alias static code analysis can be done by means of the developed library.

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

### 4.1 Ontology querying

Querying for ontologies is a well-known graph querying problem. We evaluate our library on some popular ontologies which are presented as RDF files in the paper [**?** ]. 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 [**?** ] for these graphs. The grammars for the queries are presented in Fig. 2 and Fig. 6.

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

The Meerkat-based and the GLL-based [**?** ] 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.

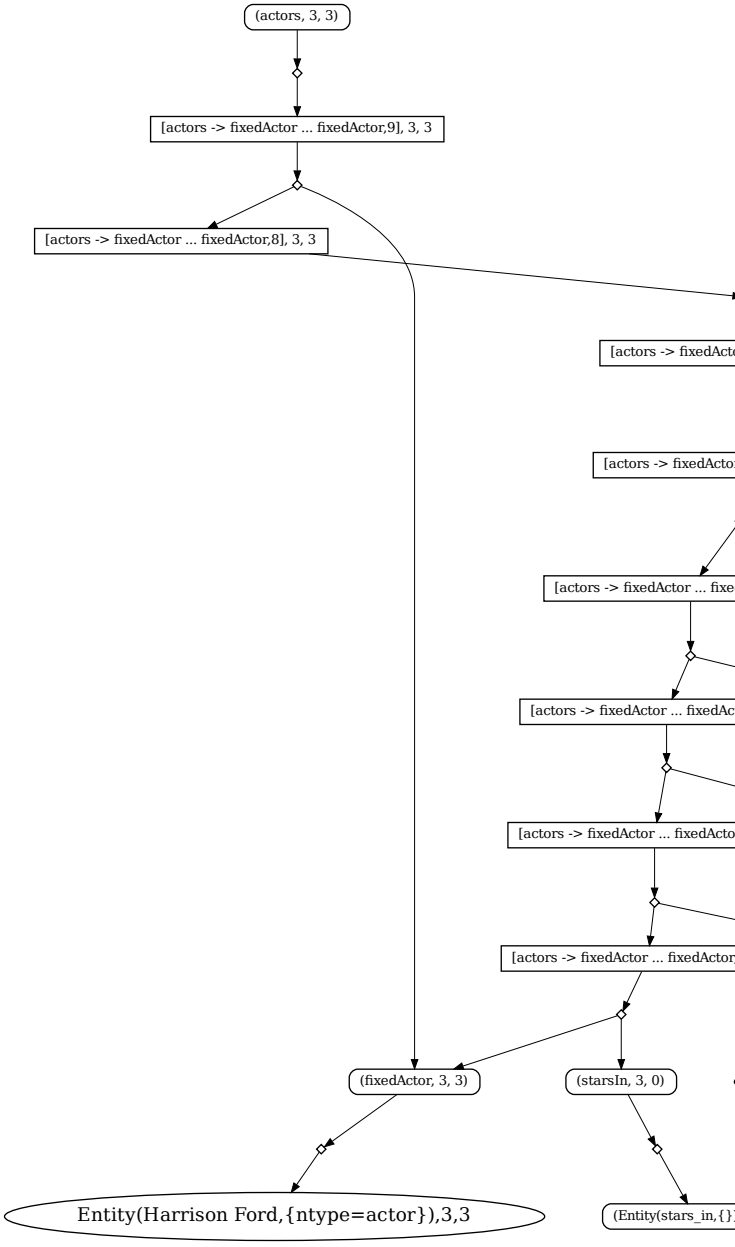


**Figure 9: SPPF: result of applying actor/movie query to the graph 4**

### 4.2 Static code analysis

Alias analysis is a fundamental static analysis problem [**?** ]: it checks may-alias relations between code expressions and can be formulated as a context-free language (CFL) reachability problem [**?** ] which is closely related to context-free path querying problem. In this analysis, a program is represented as a Program Expression Graph

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

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

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

**Figure 10: Query 2 grammar**

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

**Figure 11: Context-Free grammar for the may-alias problem**

(PEG) [**?** ]. 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 [**?** ] is presented in Fig. **??**. 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 **??**. We can conclude that our solution is expressive and performant enough to be used for static analysis problems which can be expressed as CFPQs.

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

**Figure 12: Meerkat representation of may-alias problem grammar**

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

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

## 4.3 Comparison with Trails

Trails [**?** ] 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 **??**. Trails and Meerkat-based solution compute the same results, but Trails is up to 10 times slower than Meerkat-based solution.

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

# 5 CONCLUSION

We propose a native way to integrate a language for context-free path querying into a general-purpose programming language. Our

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

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

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 [**?** ] reachability, but not context-free [**?** ]. 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 [**?** ] in te library and investigate nature of approximation.

All these improvements may provide !!! Declarative code analysis tools and languages Flix [**?** ]. Reps [**?** ]

Improved version of OpenCypher [**?** ], 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.