On Combinators and Single Source Context-Free Path Querying

Mikhail Nikilukin
Inria Paris-Rocquencourt
Rocquencourt, France
trovato@corporation.com

Ekaterina Verbitskaia
The Thørväld Group
Hekla, Iceland
larst@affiliation.org

Semyon Grigorev
Rajiv Gandhi University
Doimukh, Arunachal Pradesh, India
larst@affiliation.org

ABSTRACT

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS CONCEPTS

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

KEYWORDS

Graph Database, Context-Free Path Querying, Parser Combinators, Single-Source Path Querying, CFPQ, Language Constarined Path Querying

ACM Reference Format:
Mikhail Nikilukin, Ekaterina Verbitskaia, and Semyon Grigorev. 2018. On Combinators and Single Source Context-Free Path Querying. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Context-Free Path Querying (CFPQ) is an actively developed area in graph database analysis. CFPQ is widely used for static code analysis [?], RDF querying [?], biological data analysis [?].

Most of research focus on developping algorithms for CFPQ evaluation [?], whereas specification languages for support context-free are not investigated enough. Best to our knowledge, only one extension for Sparql supports context-free constarints: CfSparql [?]. There is also a proposal for CFPQ as a part of Cypher[1] language, but there is no implementation for it yet. We believe that more research should be conducted on the specification languages fo context-free constraints in graph querying.

It is worth noting that graph analysis is often only a part of a more complex system, usually implemented in a general-purpose language. Since a graph query language is unsuitable to implement a whole system, there should be means of integration of them into general-purpose programming languages. There are many ways to integrate them ranging from creating graph queries from string values of a general-purpose language [?] to implementing a special embedded domain specific language [?] to more sophisticated.

Although simple, the string manipulating approach does not provide a developper with any safety guarantees. There is no way to ensure that a string generated by an application is a valid query or, in case it is not, to provide any feedback. This makes string manipulating technique error prone, the code — unclear and hard to maintain.

Safety of an embedded DSL entirely depends on its implementation. Some general-purpose languages with powerfull type systems (such as Haskell, OCaml or Scala) or the ones supporting hygienic macros (such as Scheme or Rust) facilitate creating safe and reliable DSLs. Still, they typically lack full support of a development environment: it may be harder to debug queries or issues with composability can arise.

There is a general trend towards imposing more restricting type systems on programming languages. Among many others are typing annotations for Python and TypeScript code and nullability checks in Kotlin. Typing graphs and query languages improves readability and simplifies maintainance [4].

Parser combinators are the answer to the integration of parsing into a general-purpose programming language. Recursive descend parsers are encoded as functions of the host language, while grammar constructions such as sequencing and choice are implemented as higher-order functions. This idea was first introduced in [1] and further developped in numerous works. Notable development is monadic parser combinators [2]. In this approach, one can not only parse the input, but simultaneously run semantics calculation if parsing succeeds. Paper [3] proposed the first monadic parser combinator library which solves the long-standing problem of inability to handle ambiguous and left-recursive grammars. The authors earlier presented a library for graph querying was developped [5] based on this work. The core idea is to use generalized parser combinators as both a way to formulate a query and to execute it. This approach inherits benefits of combinatory parsing: ease of code reuse, type safety guaranteed by the host language and, since the parser is simply a function, the integrated development support.

Besides integration, it is also capable to compute both the single source and all pairs semantics, as well as execute user actions. The single source semantics is relevant to many real-world application, including manual data analysis. It also may be less time-intensive, since on average it needs to expore only a subgraph of the input graph. Many querying algorithms are only capable to compute all pairs reachability which makes them unsuitable for some applications.

[1] !!!

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

In this paper we make the following contributions.

- We demonstrate how to use combinatory-based graph querying on example.
- We illustrate such features of the approach as type-safety, flexibility (composability and generics), IDE support and computing user-defined actions.
- We evaluate single source context-free path querying on some real-world RDFs.
  - Based on our evaluation, the most common case in RDF context-free querying is when the number of paths in the answer set is big, but they are small.
  - We demonstrate that the single-source CFPQ can feasibly be used to evaluate such queries.
  - We conclude that there is a need to further detailed analysis of both theoretical time and space complexity.

2 EXAMPLE OF CFPQ WITH COMBINATORS

In this section we demonstrate main features of combinators in the context of context-free path querying and integration with general-purpose programming languages. To do it we first introduce a simple graph analysis problem and then show how to solve it by using parser combinators. In our wirk we use Merrkst.Graph combinators library.

Problem statement. Suppose we have an RDF graph and whatn to analize hierarchical dependencies over different types of relations. Our goal is for fhe given object to find all objects which lies on the same lavel of hierarchy. Namely, for the given set of relations $r = \{R_0 \dots R_i\}$ and for the given vertex v we want to find all vertices reachable form v by paths which specified by the following context free grammar in EBNF: $qSameGen \rightarrow {R_0}^{-1} qSameGen?\, R_0 \mid \dots \mid {R_i}^{-1} qSameGen?\, R_i$. Additionally we want to calculate lenght of these paths.

The first step is to specify paths constraint. For example, we fix relation to be `skos__narrowerTransitive`. Then constraint may be specified in terms of combinators as follows:

```
val rName = "skos__narrowerTransitive"
def qSameGen () =
    syn(inE((_: Entity).label() == rName) ~ qSameGen().? ~
        outE((_: Entity).label() == rName))
```

Here we use `inE` and `outE` to specify incoming nad outgoing edges with respective label, `~` to concatenate subqueryes, and `.?` to specify zero or one repitition of subquery.

This query specifies exactly path we want, but still not a solution. First of all, we can not specify start vertex and can not extract final vertices. Also, this query is for one specified relation. To investigate hierarchy over a set of relations we need to rewrite it.

Compositionality. First step is a generalization of the query to simplfy handling of different types of relations. To do it we introduce a helper function `reduceChoice` which takes a list of subqueryes and combine them by using alternation operation.

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

After that we use this function in new version of `sameGen` to combine subqueryes for different types of braces. To make it possible to use differetn types of braces without query rewriting we pass barces as a parameter.

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

Now we are ready to provide ability to specify start vertex and collect information of final vertices. First of all, we provide a filter to select only vertices with `uri` property.

```
val uriV = syn(V((_: Entity).hasProperty("uri")) ^^)
```

After thet we create a function which takes two parameters, start vertex and a path query, and create a new query to find all vertices with `uri` property which are reachcbl from the specified strat vertex by specified path. Finally we collect values of `uri` for all reachable vertices. To do it we specify user-defined action `{case _ ~ _ ~ (v: Entity) => v.getProperty[String]("uri")}` which captures result of query (it is a triple-sequence of subqueryes results) and gets the `uri` property form result of last subquery.

```
def queryFromV (startV, query) =
  syn(startV ~ query ~ uriV &
      {case _ ~ _ ~ (v: Entity) =>
           v.getProperty[String]("uri")})
```

User-defined actions. Final step is to extend the query with calculation of lengths of all paths which satisfie conditions. To do it we equip `sameGen` with additional user-defined actions.

```
def sameGen(brs: List[(_,_)]) =
    reduceChoice(
      brs.map {
        case (lbr, rbr) =>
          syn((lbr ~ (sameGen(brs).?) ~ rbr) & {
            case _~Nil~_ => 2
            case _~((x:Int)::Nil)~_ =>  x + 2
          })
      })
```

The `queryFromV` now handles not only theard element, but also the second one in order to get access to accamulated lenghts.

```
def queryFromV(startV, query) =
    syn(startV ~ query ~ uriV &
      {case _ ~ (len:Int) ~ (v:Entity) =>
        (len, v.getProperty[String]("uri"))})
```

Now we are ready to bring all functions togeteher and evaluate the uery. To do it first we add a helper function `makeBrs` which takes a list of relation names and create list of pairs of subqueryes which check incoming and outgoung respectively (a pairs of brackets).

```
def makeBrs (brs:List[_]) =
    brs.map(name =>
       (syn(inE((_: Entity).label() == name) ^^),
        syn(outE((_: Entity).label() == name) ^^)))
    .toList
```

We use this function in the main function `runExample` which takes a list of relations, start vertex and the graph, build the same generation query over given relations by useing specified functions, and execute this query form the given vertex for the given graph.

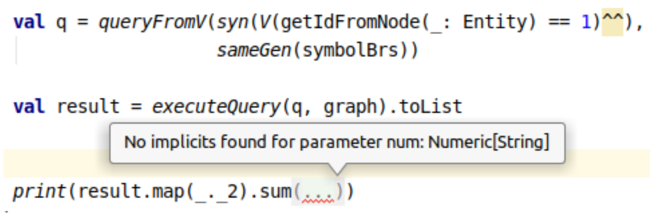
```
def runExample (brs: List[_], startVId, graph) =
    val startV = V(getIdFromNode(_: Entity) == startVId
    executeQuery(queryFromV( syn(startV)^^),
                    sameGen(makeBrs(brs))),
            graph).toList
```

Finally, to execute the query wot we want from the vertex 1 we should call `runExample` as presented below.

```
runExample(RdfConstants.RDFS__SUB_CLASS_OF :: Nil, 1, graph)
```

Type safety. As far as queryes are exxpressed in terns of functions of general purpose language which you use for target application development, compiler provide static type checking of queryes and its results.

In the example showed in figure 1, elements of pair wich represents query result are used incorrectly: we want to find total length of all paths but sum final vertices' identifiers instead of lenghths. As a result, compiler statically detect a error because integer expected instead of string.



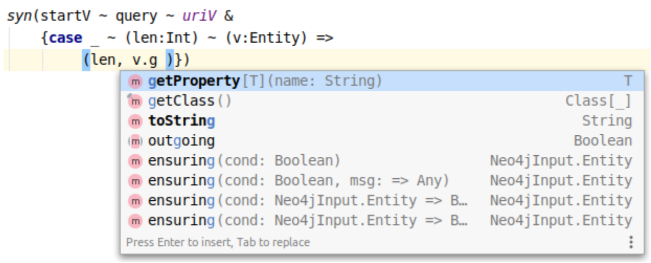
```
val q = queryFromV(syn(V(getIdFromNode(_: Entity) == 1)^^),
                   sameGen(symbolBrs))

val result = executeQuery(q, graph).toList
               No implicits found for parameter num: Numeric[String]

print(result.map(_._2).sum(...))
```

Figure 1: !!!

IDE Support. Since you can use IDE for development, you get all features for qury development, such as syntah highkighting, code navigation, autocompletion, without any additional effor. An example of autocompletion suggestions for vertex is presented in figure 2.



```
syn(startV ~ query ~ uriV &
    {case _ ~ (len:Int) ~ (v:Entity) =>
        (len, v.g )})
          getProperty[T](name: String)                T
          getClass()                           Class[_]
          toString                               String
          outgoing                              Boolean
          ensuring(cond: Boolean)         Neo4jInput.Entity
          ensuring(cond: Boolean, msg: => Any)  Neo4jInput.Entity
          ensuring(cond: Neo4jInput.Entity => B…  Neo4jInput.Entity
          ensuring(cond: Neo4jInput.Entity => B…  Neo4jInput.Entity
          Press Enter to insert, Tab to replace
```

Figure 2: !!!!

3 EVALUATION

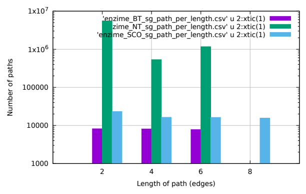
We evaluate Meerkat.Graph on single source context-free path querying scenario. For evaluation we use Neo4j graph databese which was run on PC with the following configuration: CPU, RAM, OS, JVM.

Neo4j is integreted into application !!!!

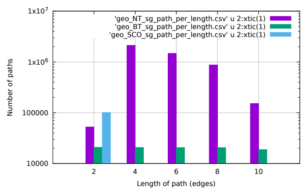
Dataset contains two real-world RDFs: Geospecies which contains information about biological hierrarchy[2] and Enzime which is a part of UniProt database[3]. Detailed description of these graphs

[2]https://old.datahub.io/dataset/geospecies. Access date: 12.11.2019.

[3]Protein sequences data base: https://www.uniprot.org/. RDFs with data are available here: ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf. Access date: 12.11.2019



(a) Enzime



(b) Geospecies

Figure 3: Paths length destribution

is presented in table 1. Note, that graphs was loaded into database fully, not only edges which laballed by relations used inqueryes.

Graph	#V	#E	#SCO	#T	#NT	#BT
Enzime						
Geospecies						

Table 1: Details of graphs

Queries for evaluation are versions of same-generation query — classical context-free query which is useful for hierarchy analysis. All queryes in our evaluation a created by using functions which described in the section ??. Namely we create and evaluate three queries Q_1, Q_2 and Q)3 as presented below.

```
def q1 (startV) =
   val q =
      sameGen(makeBrs(RDFS__SUB_CLASS_OF ::
                   RDF__TYPE :: Nil))
   queryFromV(startV, q)

def q2 (startV) =
   val q =
      sameGen(makeBrs(skos__narrowerTransitive :: Nil))
   queryFromV(startV, q)

def q3 (startV) =
   val q =
      sameGen(makeBrs(skos__narrowerTransitive :: Nil))
   queryFromV(startV, q)
```

As you can see, once create a set of appropriate functions, one can easely construct new queryes.

For each graph and each query we run this query form each vertex from graph and measure elapsed time and required memory by using !!! tool. Note, that mesured memory is allocated by JVM, not really used.

Enzime RDF querying. .

Results of evaluation are presented in figures 4 and 5. Also we collect paths length distribution which is showed in figure 3. We can see that prvided datasets contain relatively short paths which satisfie queryes.

Figure 4 shows dependency of query evaluation time on query answer size in terms of number of edge-different !!! paths. First of all, we can see that evaluation time is linear on answer size. Also we can see, that time which requred to evaluate query for one specific vertex is relatively small. In our case it is less than 90ms.

Figure 5 shows dependency of memory requred to evaluate qurey on query answer size in terms of number of uniqie paths.

Geospecies RDF querying.

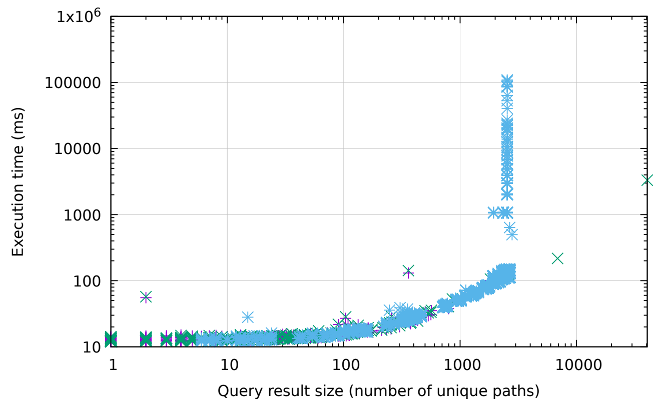


Figure 4: Query execution time for Enzime dataset and queryes Q_1 and Q_2

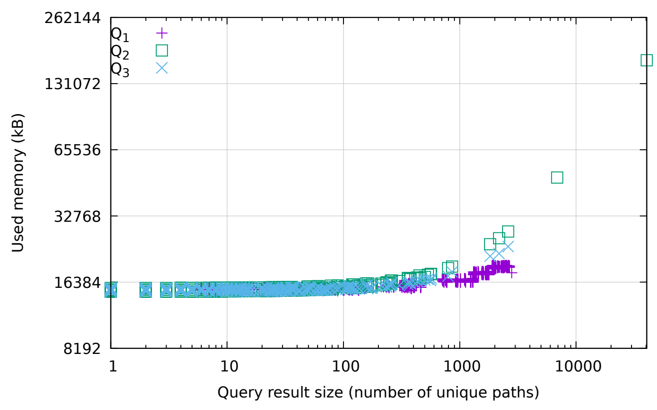


Figure 5: Query required memory for Enzime dataset and queryes Q_1 and Q_2

Here we can see !!!!

Finally, we can conclude that confext-free path querying in single source scenario can be efficiently evaluated by using !!! in case when number of paths in answer is big but its length is relatively small. While all pairs scenario is still hard [?], single source scenarion, which is useful for manual or interactive data analysis, can be !!! Also we can see that while theoretical time and space complexity of CFPQ algoritms at leas cubic, in demonstrated scenario real execution time and required memory is linear. So, it is necessary to provide detailed time and space complexity analysis of algorithms.

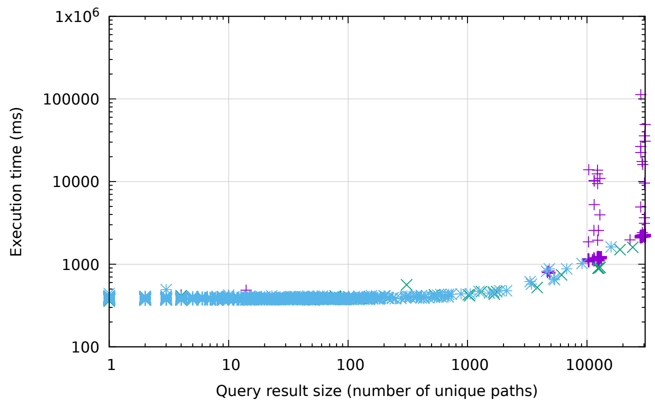


Figure 6: Query execution time for Enzime dataset and queryes Q_3 and Q_4

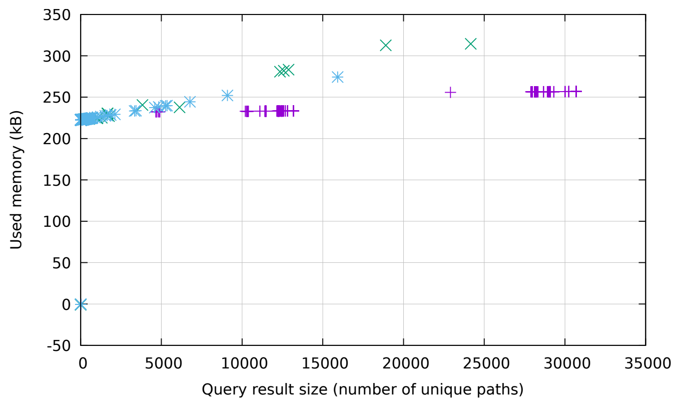


Figure 7: Query execution time for Enzime dataset and queryes Q_3 and Q_4

4 CONCLUSION AND FUTURE WORK

We show that single-source context-free path querying can be !!! We demonstrate a combinator-based approach implemented in Meerkat.Graph Scala library, but this approach can be implemented in almost any high-level programing language. While combinators is a very powerful way to specify context-free queries, it may seem hard to understand for many users. There are other algorithms for context-free path queries which should be applicable for single-source path querying and we hope that they can be integrated with the existing graph database in a more convenient way. But it is necessary more research in this direction.

We should investigate wore datasets to detect other shapes of query results. For example, we should investigate the behavior of single-source querying in the case when a number of resulting paths is small, but paths are relatively long. And the first question is which data analysis tasks lead to this scenario.

One of important direction of the future reserach is to optimize performance of proposed solution. One of possible solution is deep integration with Neo4j infrastructure to utilize cache system.

Another direction is combinators library improvement. First of all, it is necessary to make cimbinators syntax more user-friendly. Also, it is necessary to create set of query templates (see same-generation template).

ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

REFERENCES

[1] William Burge. [n.d.]. Recursive Programming Techniques.

[2] Graham Hutton and Erik Meijer. 1996. Monadic parser combinators. (1996).

[3] 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*. 1–12.

[4] Norbert Tausch, Michael Philippsen, and Josef Adersberger. 2011. A Statically Typed Query Language for Property Graphs. In *Proceedings of the 15th Symposium on International Database Engineering & Applications* (Lisboa, Portugal) *(IDEAS '11)*. Association for Computing Machinery, New York, NY, USA, 219–225. https://doi.org/10.1145/2076623.2076653

[5] 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 Symposium on Scala* (St. Louis, MO, USA) *(Scala 2018)*. Association for Computing Machinery, New York, NY, USA, 13–23. https://doi.org/10.1145/3241653.3241655