Querying Relational Databases Using Python-Integrated DSL

MATT KOTZBAUER, Harvard University, USA

For my CS252R final project, I worked under Will Byrd and the UAB lab team to create a DSL that can run queries through biomedical reasoning graphs by generating mediKanren queries from Python. To do this, I created a series of functions in Python that define the queries as a class, and created operations on this class that allow the user to assemble mediKanren queries from short Python declarations, write and run queries from Racket files, and parse the results of the queries into Python lists. This also allows for an intuitive way to assemble multi-hop queries, where the results of independent queries can be referenced within Python. All referenced code can be found on my GitHub!

Additional Key Words and Phrases: miniKanren, Domain-Specific Languages, Racket, Python

1 INTRODUCTION

Below is an example of a mediKanren query within a biomedical reasoning database:

```
(define regulates-EGFR
  (time (query:X->Known
         #f
         (set->list
          (get-non-deprecated-mixed-ins-and-descendent-predicates
             *-in-db
           '("biolink:regulates")))
         (set->list
          (get-descendent-curies*-in-db
           (curies->synonyms-in-db (list "HGNC:3236")))))))
```

Though mediKanren is a very expressive query language, a number of UAB lab team members don't have experience using it, noting that Python may be easier to use when making queries. What if this query could instead become the below Python code?

```
# Initialize member of query class
sample_query = Query(type="X->Known", subject=ALL, predicate="
    biolink:regulates", object="HGNC:3236")

# Assemble query string based on class member
sample_default_query = sample_query.assemble_query()

# Write query to file and run it
query_output = write_and_run(~/mediKanren/query.rkt,
    sample_default_query)
```

My goal in this project was to implement a simple, easily-learnable domain-specific language in Python that would allow members of the lab team to interact with the biomedical reasoning databases - writing, running, and parsing queries. The end result is a Python query class and series of helper functions that allow for this.

2 PROBLEM

Biomedical reasoning graphs consist of millions of nodes and edges signifying complex relationship between biological factors, allowing for robust querying applications. The lab team's current mediKanren implementation allows them to run full queries on these graphs within seconds - the goal of my DSL is to generate the most common types of queries so that more people can create and run these scripts.

3 DSL STEP-THROUGH

To build intuition, we will first go through a demonstration of how we can run mediKanren queries in Python using the DSL. To start the DSL, we simplify queries into 4 elements: their type, subject, predicate, and object.

```
query ::= (type, subject, predicate, object)
```

Where for each subpart, we have types and predicates defined as strings, and subjects and objects being allowed to be either a string or a list of them:

```
type, predicate ::= string
subject, object ::= string | list[string]
```

We declare queries as a Python class and can use the assemble_query() helper function to assemble a string from its current state:

```
sample_query = Query(type="Known->X", subject="PUBCHEM.COMPOUND
    :5291", predicate="biolink:regulates", object=ALL)
query_string = assemble_query(sample_query)
```

These queries are generated from templates of existing mediKanren queries, deciding how to plug in the values based on the query type (X->Known, Known->X, X->Known->Faster, etc.). Once we have the string denoting the query, we can either A) write it to a .rkt file and reference it in an interactive Racket session, or B) run the .rkt file using the write_and_run helper function:

```
Case A (interactive usage):
    (Python):
        write(file_path/"query.rkt", query_string)
    (Racket):
        ,en "query.rkt"
Case B (static usage):
    (Python):
        write_and_run(file_path/"query.rkt", query_string)
```

Case A has the benefit of being able to be loaded in an interactive environment, as the ",en" command within the Racket session will allow for changes in the file to be loaded, and therefore allow the user to write and run multiple Python scripts flexibly within the same session. Case B can be directly loaded from Python, but each running script needs to re-load the reasoning graph from scratch, which for most computers takes 5-10 minutes. This makes pure-Python implementations better-suited for longer sequences of queries. We can now run independent queries within the Python DSL implementation.

In order to parse these queries back into the table, we can print the result of the query in the terminal and parse using a helper function:

```
formulated_output = table_output(query_output)
```

This creates a 2D list representation of our query output. Within each row, the first 3 elements are the subject, predicate, and object of the query result. The fourth and final element is a dictionary for related attached information, which maps the title as a key and the list of according information as its value. With this, we capture the query output within the table, and can now reference the results of our queries.

For a faster query, we can use the assemble_fast_query() function, which draws from a template in which we categorize regions of data into buckets to speed them up:

```
fast_query = Query(type="Known->X", subject="PUBCHEM.COMPOUND
    :5291", predicate="biolink:regulates", object=ALL)
fast_query_string = assemble_fast_query(fast_query)
```

The rest of the implementation (writing it to the file and running in Racket / Python) is then identical.

To hop between queries, the Python DSL runs the queries independently in mediKanren, and then connects the results by noting similarities between the relevant subject and object of the resulting tables. An example would be as follows:

```
query_1 = Query(type="X->Known", subject=ALL, predicate="biolink:
    regulates", object="HGNC:3236")
query_2 = Query(type="Known->X", subject="PUBCHEM.COMPOUND:5291",
     predicate="biolink:regulates", object=ALL)
sample_hop = query_hop(query_1, query_2)
```

The results of the hop are then in table form. If we want to define the hop in terms of table results, we could also do that through the following using table_hop, which behaves identically to query_hop but operates on query-result tables rather than members of the Query class:

```
table_1 = table_output(write_and_run(query_1))
table_2 = table_output(write_and_run(query_2))
sample_hop = table_hop(table_1, table_2)
```

Both blocks of code produce an identical result - the hop between queries 1 and 2. The second block of code can be used to assemble multi-hop queries, where the results of hops are expressed as tables. An example would be as follows:

```
table_3 = table_output(write_and_run(Query(type = "Known->X",
    subject=ALL, predicate="biolink:regulates", object="PUBCHEM.
    COMPOUND:2244")))
multi_hop = table_hop(table_3, sample_hop)
```

We can now use the DSL to make multi-hop queries, and this concludes the description of the main helper functions! We've built up a DSL from which we can generate and run mediKanren queries, capture their output in table form, and use this form to create hops and multi-hops between queries.

4 DSL DESIGN

Let's now go through the descriptions and typing rules for each of the main helper functions. Firstly, we have the helpers to assemble strings from members of the query, assigning a template based on the query's type. For values marked as unknown (which can only be the subject / object), we type-check with an auxiliary function to allow for multiple curies to be searched (in the case of a list, it converts a Python list of strings into a Racket list, and in the case of a string it converts it to a Racket list with a single element). We then define the following string-assembly functions, where the returned string is the Racket query that's been assembled:

```
# Default assembly of query (as seen in regulates-EGFR, diabetes-
    causes, and diabetes-treatments)
assemble_query(input_query: Query) -> str:

# Assembly of query that uses buckets to be faster (as seen in
    regulates-EGFR-faster and imatinib-regulates-faster)
assemble_fast_query(input_query: Query) -> str:

# Assembly of minimal-syntax query
assemble_base_query(input_query: Query) -> str:
```

We can then run our assembled queries using the below series of commands, which write our string to a Racket file and then run it. Our run() and write_run() functions also capture the query result as printed in the terminal, which is the string returned by either function.

```
# Writes string as Racket script to file path
write(file_path: str, write_string: str) -> void

# Runs Racket script specified by file path
run(file_path: str) -> str

# Combination of write and run functions: writes string as Racket
     script to file path and runs it
write_and_run(file_path: str, write_string: str) -> str
```

In order to parse our output into the two-dimensional list format, we take the string captured by the prior commands and run it through the below parsing function. The function uses the parenthese structure to categorize it into our desired table form.

```
# Translate query output string to table by parsing in Python
table_output(terminal_output: str) -> List[List[Any]]

# Resulting list form: [l1, ..., ln], where li = [subject,
    predicate, object, {key_1: aux_list_1, ..., key_m: aux_list_m
    }]
```

We also have our hop functions, which allow us to define hops either from queries themselves (in which case the strings are generated, scripts executed, results parsed, and tables 'hopped' together), or from the table-results of queries.

```
# Defines a hop between two queries
query_hop(query1: Query, query2: Query, file_path: str) -> List[
    List[Any]]

# Defines a hop between the table results of two queries
table_hop(table1: List[List[Any]], table2: List[List[Any]]) ->
    List[List[Any]]
```

That's the last of the helper functions! All of these functions are stored in helpers.py of the GitHub project.

5 DSL DEMO

For a demonstration, we can display a couple of queries and results generated by the DSL.

Example 1: Standard Query

We can define the following base case query within our Python DSL, seeking to find factors that regulate the expression of the EGFR gene (which has HGNC identifier 3236, as referenced in the object of the query):

```
std_query = Query(type="X->Known", subject=ALL, predicate="
    biolink:regulates", object="HGNC:3236")
query_string = assemble_query(std_query)
result_string = write_and_run("~/sample.rkt", std_query)
result_table = table_output(result_string)
```

We find that query_string, the assembled Racket query, becomes

```
(time (query:X->Known
            #f
            (set->list
            (get-non-deprecated-mixed-ins-and-descendent-
                predicates*-in-db
            '("biolink:regulates")))
            (set->list
            (get-descendent-curies*-in-db
            (curies->synonyms-in-db (list "HGNC:3236"))))))
```

And after running the query, the resulting terminal output shows

```
...'(("PUBCHEM.COMPOUND:4477" "biolink:regulates" "NCBIGene:1956"
     ("biolink:aggregator_knowledge_source" "(infores:hetio)") ("
    biolink:primary_knowledge_source" "infores:lincs") ("object" "
    NCBIGene:1956") ("object_direction_qualifier" "upregulated")
    ("predicate" "biolink:regulates") ("subject" "PUBCHEM.COMPOUND
    :4477"))...
```

Finally, after running table_output, the resulting 2D list becomes

```
[['PUBCHEM.COMPOUND:4477', 'biolink:regulates', 'NCBIGene:1956',
    {'biolink:primary_knowledge_source': ['infores:lincs'], '
    object': ['NCBIGene:1956'], 'object_direction_qualifier': ['
    upregulated'], 'predicate': ['biolink:regulates'], 'subject':
    ['PUBCHEM.COMPOUND:4477']}], ['MONDO:0007254', 'biolink:
    regulates', 'NCBIGene:1956', ...
```

We now have a list of factors that regulate the expression of the EGFR gene.

Example 2: Query Hop

We start with Known->X and X->Known type-queries, seeking to find elements that adjoin them. In this case, we have digitoxin (PubChem ID 441207) as the subject of the first query, and EGFR as the object of the second. This gives us the following queries:

```
sample_query1 = Query(type="Known->X", subject="PUBCHEM.COMPOUND
    :441207", predicate="biolink:regulates", object=ALL)

sample_query2 = Query(type="X->Known", subject=ALL, predicate="
    biolink:regulates", object="HGNC:3236")
```

The second query is identical in structure to the one from Example 1, but the first takes the following structure:

```
(time (query:Known->X
            (set->list
            (get-descendent-curies*-in-db
            (curies->synonyms-in-db (list "PUBCHEM.COMPOUND
                :441207"))))
            (set->list
            (get-non-deprecated-mixed-ins-and-descendent-
                predicates*-in-db
            '("biolink:regulates")))
            #f))
```

Let's now define a hop between queries 1 and 2:

```
hop_result = query_hop(sample_query1, sample_query2)
```

The result of the hop becomes the following table:

```
[[['PUBCHEM.COMPOUND:441207', 'biolink:regulates', 'NCBIGene
    :8802', {'biolink:primary_knowledge_source': ['infores:lincs
    '], 'object': ['NCBIGene:8802'], 'object_direction_qualifier':
     ['downregulated'], 'predicate': ['biolink:regulates'], '
    subject': ['PUBCHEM.COMPOUND:441207']}], ['NCBIGene:8802', '
    biolink:regulates', 'NCBIGene:1956', {'biolink:
    primary_knowledge_source': ['infores:lincs'], 'object': ['
    NCBIGene:1956'], 'predicate': ['biolink:regulates'], 'subject
    ': ['NCBIGene:8802']}]]]
```

From this result, we find SUCLG1 (NCBIGene:1956) to be an intermediate step between digitoxin and EGFR (though the results of this query doesn't show the direction in which SUCLG1 regulates EGFR).

6 MORE IMPLEMENTATION IDEAS

In my final time left with this project, I wanted to create a Python integration with the mychem.info API, so that we could, for example, get the PubChem ID results from a query, and compare their smiles values to compare their relative shapes using a library like RDKit. I unfortunately wasn't able to hammer out bugs with the API, but this could be a cool way to find similarities in the shapes of treatment chemicals.

Further, we could set up a local server to automatically feed generated queries into the Racket session, the benefit being that the user wouldn't have to re-enter the ",en" Racket command upon running the Python script. This would probably be the most seamless use case of the DSL, as the user may not have to have more than a single window open in order to run it.

7 RELATED WORK

For further understanding of the scripts that are being generated by the DSL, the original mediKanren paper can be a good background for its use cases and structure, as well as the logistics as the knowledge graphs themselves.

I wasn't able to find papers that were a strong match with what I was doing, but this paper from a few months ago explores how simplifying highly-structured languages to BNF grammar-based domain specific languages can improve LLM performance in synthesizing solutions. This also somewhat relates to the library learning paper that I presented in class at the start of the semester, where intermediate abstractions are made from primitives in order to reduce the search space.

The idea of creating a minimal DSL somewhat relates to what we did on the first homework as well, where some implementations allowed you to reduce the number of available operations in order to decrease the search space - though the attempted simplicity of my DSL is moreso for user experience, and my project doesn't iterate through a search space to generate its queries (instead applying templates based on query type).

8 FUTURE WORK / CONCLUSION

I had a lot of fun working on this project, and am definitely aiming to continue working on it in the future. I think the most immediate next step would be to try and implement the interactive server so that the experience of creating and running queries is more smooth. I also want to explore more bioinformatics pipelines that can be assembled from the query language, as there seem to be a number of cool directions that could go (LLM's, generating visualizations of query results, comparing protein structure).

ACKNOWLEDGMENTS

You've made it to the end - thank you for reading! Huge thanks to Will Byrd, Professor Amin, and my friends Kento and Chris for helping me with ideas for implementation, and getting me un-stuck many times over the course of this project. This was my first time doing a school project of this scale, and the advice really helped me to produce results that I'm happy with.