A Coq Mechanized Formal Semantics of Cypher

Svetlana Semenova

HACS287

Faculty Sponsor: Leonidas Lampropoulos

Abstract

Cypher is a language used for querying and updating property graph databases, which has increased in popularity with the rise of commercial graph databases. This has created the need for mechanized, formal semantics for Cypher queries in a proof assistant like Coq. A semantics of Cypher has already been written out; this project aimed to translate those semantics into Coq. Much of the syntax and semantics were formalized successfully, with space for future developers to implement and test various matching algorithms.

Introduction

Cypher is a language used for querying and updating property graph databases. It was originally developed by Neo4j for their implementation of a property graph database,[1] but since then, it has been adopted, modified, and used by other commercial graph databases, such as RedisGraph[2] and Memgraph,[3] among others. It even has had an impact on the GQL project, an international effort to create a standard graph query language.[4] Needless to say, Cypher is a quickly growing and influential language that will only continue to grow from here.

As a growing language with several implementations in the market, it became necessary for the community to create a formal semantics that can then, in future efforts, be used either to formally verify the correctness of a given implementation or as an oracle to test it.[5] This, in turn, can ensure the integrity of such implementations, which has historically starkly reduced miscompilations. And especially with Cypher, a language that directly interfaces with data, reducing bugs in its implementation is critical to keep the data it works with secure.

A semantics for Cypher has already been written out by Francis et. al.: a group of researchers spread between Universite Paris-Est, University of Edinburgh, and Neo4j.[6] Translating these semantics into a proof assistant, like Coq, is a reasonable next step. This will allow Cypher programs to be processed in Coq, which can then be used to formally

---

[1] "Cypher Query Language," Neo4j, accessed January 28, 2023, https://neo4j.com/developer/cypher/
[2] "RedisGraph: A Graph database built on Redis," Redis, accessed January 28, 2023, https://redis.io/docs/stack/graph/
[3] "Neo4j vs Memgraph," Memgraph, accessed January 28, 2023, https://memgraph.com/memgraph-for-neo4j-developers
[4] "GQL Standard," Graph Query Language GQL, last modified December 2, 2022, https://www.gqlstandards.org/home
[5] John Longley, "Goals of formal semantics," The University of Edinburgh School of Informatics, accessed January 28, 2023, https://www.inf.ed.ac.uk/teaching/courses/fpls/note2.pdf
[6] Nadime Francis, et al, "Formal Semantics of the Language Cypher." arXiv, March 20, 2018. http://arxiv.org/abs/1802.09984.

verify that the programs follow the specifications they claim to be following. As well, having a mechanized semantics will make it easier to formally verify implementations in the future. Therefore, my goal in this project was to do just that: mechanize the semantics written by Francis et. al. in Coq.

I first formalized the data structures Cypher uses to represent graph data: graphs, paths, and patterns. I then defined satisfaction of patterns, which led to the definition of path pattern matching. I defined the syntax of Cypher and its semantics, and created a sample graph and query as a proof-of-concept.

Methodology

The methodology for this project can be stated very succinctly: take the formalizations written by Fracis et. al. and translate them into Coq code. This involved first encoding the basic data structures Cypher needed. Then, the formal semantics were written in Coq, in essence creating an AST tree — also defined as an inductive data structure — that could then be analyzed in Coq. Writing the semantics is split between defining expressions and defining queries.

Basic Data Structures

The basic data structures needed were identifier types, paths, values, graphs, and tables.

Identifier types are property keys, node identifiers, relationship identifiers, node labels, and relationship labels. The first three were represented as an inductive data structure with one constructor that takes one natural number, and the last two were the same but with a string. The reason I chose to create a separate inductive data structure, rather than just pass around numbers or strings with type labels, is to ensure that there could be no overlap. For example, given `ID_Key 2` and `ID_Node 2`, these objects needed to be distinct. In Coq, `2:id_key` is equal to `2:id_node`, despite technically being of different types. So although this creates some overhead, it ensures correctness of equality. All identifier types got decidability theorems to allow for sets of them.

Paths were defined with a simple inductive data structure. Several supporting theorems were written: e.g. getting the length of the path, concatenating paths, getting the nth node, and so on.

Values in Francis et. al. were defined inductively, and so I defined them inductively as well. Two value types were not completed due to lack of time: integers and maps. Integers currently have natural numbers as a placeholder, as Coq does not support integers simply, and maps are currently empty. I added one extra value type, `V_Error`, to be able to distinguish when a type error occurred when evaluating expressions.

Graphs were defined as a tuple in Francis et. al., and I did the same. There are, however, a few functional differences between the detailed semantics and my implementation:

- First, ι, the function responsible for taking node/relation identifiers and keys to values got split into two: one function for taking nodes and keys to values, and one for taking relations and keys to values. This was done in order to make proving things cleaner down the line.
- Second, each graph is meant to have its own set of node labels and relationship types to draw from. However, the graphs as I have written them draw from universal types (defined as identifier types). This was done for simplicity, and could (and should be) modified once the rest of Cypher is coded properly.
- Finally, Francis et al. did not explicitly define any way of qualifying whether a specific graph is valid or not, but had some implicit assumptions, e.g. relationships point to and from nodes that exist in the graph. I wrote a proposition `valid`, which could be used down the line in proofs about operations on valid graphs. However, one assumption, explicitly stated — that ι has finite nonnull outputs — did not get added into the valid proposition, which is an error that needs to be resolved in the future.

A few simple functions to extract components of the graph were also written.

Although unused as of now, I also wrote a proposition that takes in a path and a graph and decides whether or not the path exists within the graph.

Tables are lists of records, and records are mappings from variable names to values. These were constructed simply using built-in Coq types.

Expressions

The syntax of Cypher expressions was written as an inductive data type, creating an AST Cypher tree. It involved operations like extracting elements from lists, comparison operations, and logic operations. Francis et. al. did not define syntax of arithmetic, and so those were not defined.

The semantics of expressions was defined as a function — an evaluator. Most of the semantics, including logic, value operations, and basic list operations, were completed. List selection, map operations, and full string support, were not completed fully due to time.

Queries

To define queries, I started by defining path patterns, satisfaction of path patterns, and the matching operation.

Path patterns were defined as an inductive data structure, similar to paths, of node patterns and relationship patterns. Both of those were defined as tuples, which is similar to how Francis et. al. defined them. Satisfaction of rigid path patterns given a path and record is defined as an inductive proposition, similarly to the way Francis et. Al. defined them. Satisfaction of non-rigid patterns builds on top of them. There are currently two functional differences between the official semantics and my implementation: my implementation does not support `nil len` the way Francis et. al. does, but as this is equivalent to `(1,1)`, this is not a critical semantic meaning; the second is that some things are not being properly checked, namely key existence and binding of variable names. The second is necessary for more complex queries to work correctly, and so will need to be completed.

The matching operation — which goes from a graph, path, and variable binding to a table — was defined as a proposition, despite technically needing to be a function. This was because Francis et. al. defined the end result in set-builder notation going over an infinite domain (which they commented as still necessarily resulting in a finite table). Actually computing pattern matches is incredibly difficult, and was outside of the scope of this project.

Because the core computation — matching — is not defined, the rest of the semantics of queries could not be defined as function. However, as future users of this code would likely have a specific matching algorithm they want to test, I wrote a skeleton of query evaluation, which would work only after the matching function is defined. The evaluation of `WHERE`, `WITH`, and `UNWIND` were not completed. See Appendix A for code written.

Results

As a proof of concept, I translated a graph and query from Francis et. al. into Coq.

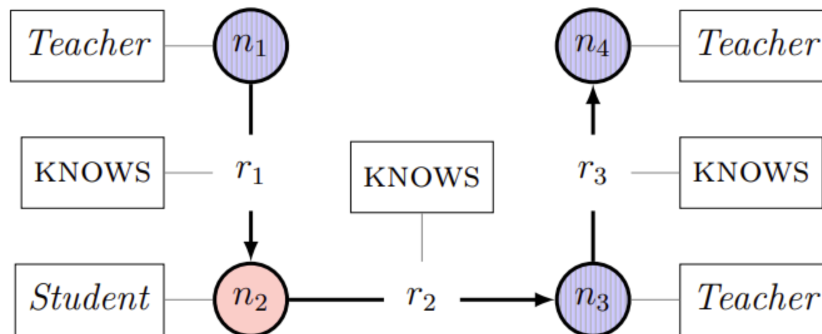

Figure 1: Reprinted from Francis et. al. p. 9

```
Definition teacher_graph : graph :=
  (
    (* Set N: nodes of G *)
    (
      set_add eq_dec_id_node (ID_Node 1)
        (set_add eq_dec_id_node (ID_Node 2)
          (set_add eq_dec_id_node (ID_Node 3)
            (set_add eq_dec_id_node (ID_Node 4) nil)))
    ),

    (* Set R: relationships of G *)
    (
      set_add eq_dec_id_rel (ID_Rel 1)
        (set_add eq_dec_id_rel (ID_Rel 2)
          (set_add eq_dec_id_rel (ID_Rel 3) nil))
    ),

    (* src: R->N, maps relationship to source node *)
    (fun x:id_rel => if eq_dec_id_rel x (ID_Rel 1) then Some (ID_Node 1) else
                    (if eq_dec_id_rel x (ID_Rel 2) then Some (ID_Node 2) else
                    (if eq_dec_id_rel x (ID_Rel 3) then Some (ID_Node 3) else None))),

    (* tgt: R->N, maps relationship to target node *)
    (fun x:id_rel => if eq_dec_id_rel x (ID_Rel 1) then Some (ID_Node 2) else
                    (if eq_dec_id_rel x (ID_Rel 2) then Some (ID_Node 3) else
                    (if eq_dec_id_rel x (ID_Rel 3) then Some (ID_Node 4) else None))),

    (* Extra information: none for this graph *)
    (fun x:id_node => fun y:id_key => None:option value),
    (fun x:id_rel => fun y:id_key => None:option value),

    (* λ: id_labels for nodes *)
    (fun x:id_node => if eq_dec_id_node x (ID_Node 1) then (set_add eq_dec_id_label (ID_Label "Teacher") nil) else
                     (if eq_dec_id_node x (ID_Node 2) then (set_add eq_dec_id_label (ID_Label "Student") nil) else
                     (if eq_dec_id_node x (ID_Node 3) then (set_add eq_dec_id_label (ID_Label "Teacher") nil) else
                     (if eq_dec_id_node x (ID_Node 3) then (set_add eq_dec_id_label (ID_Label "Teacher") nil) else
                      (nil:set id_label))))),

    (* τ: id_reltype for id_rels *)
    (fun x:id_rel => if eq_dec_id_rel x (ID_Rel 1) then Some (ID_Reltype "KNOWS") else
                    (if eq_dec_id_rel x (ID_Rel 2) then Some (ID_Reltype "KNOWS") else
                    (if eq_dec_id_rel x (ID_Rel 3) then Some (ID_Reltype "KNOWS") else None)))
).
```

Figure 2: Translated graph with students and teachers

This code could have been made shorter with Coq syntax extension, specifically for sets and for the mappings, which are essentially partial maps. However, doing it this way ensured no ambiguity, which was especially useful during construction. Notation will need to be added for ease of use once a semantics evaluator is deemed complete.

Although evaluating a query is currently impossible, it is possible to formally write them out. The following query:

```
MATCH (x:Teacher) -[:KNOWS*2]-> (y) RETURN *
```

sampled from Francis et. al. p. 10, translates to the following code:

```
Definition sample_path : path_pattern :=
  Conn
    (* (x) *)
    ((Some (Name (("x"%string):var_name)),(nil: set id_label),(fun y:id_key => None:option value)))
    (* -[:KNOWS1..3]-> *)
    ((left_to_right),(None:option id_node),(set_add eq_dec_id_reltype (ID_Reltype "KNOWS") nil),
      (fun y:id_key => None:option value),(None,Some 2))
    (* (y) *)
    (Sing ((Some (Name (("y"%string):var_name))),(nil: set id_label),(fun y:id_key => None:option value)))
.

Definition sample_query : query :=
             (* MATCH path_pattern             RETURN * *)
  Simple (CQ (MATCH (OnePatTup sample_path)) (RETURN Star)).
```

Figure 3: Translated query

As with the graph, if Coq syntax extension had been used, this could have been written in a shorter fashion, but this ensures no ambiguity about the meaning of the query.

Conclusion

Having a Coq mechanized semantics for Cypher will allow future projects to have a formalization to rely on. Specifically, any projects that require some way to formally interface with graphs can rely on this formalization. For example, Java holds its objects as a graph, and the garbage collector runs on said graph when looking for inaccessible objects. Having a way to formally interface with the graph can allow researchers to prove the correctness of specific garbage collection implementations.

However, this formalization isn't complete. For one, the details of the matching implementation are missing due to the way they're presented in Francis et. al. Without that algorithm, the rest of the query evaluation process is unable to run. However, a skeleton is built around that placeholder, so future users can fill it in to their needs and then use the formalization as necessary. As well, certain aspects of the semantics — parts of expression, specific types of clauses — are incomplete due to time constraints of the project.

There are many other things to do before this formalization can be considered complete. Beyond simply completing the semantics, there are many useful lemmas that should be least written down. For example, base ones like the closure of matching need to be established. More complex ones — e.g. proving that a non-matching node won't be in the result — should also be written out. Then, after the matching implementation is filled in, these lemmas can be used to check the correctness of that implementation. Finding a set of lemmas that completely describe the behavior of matching and other operations is a worthwhile future endeavor.

References

Francis, Nadime, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, et al. "Formal Semantics of the Language Cypher." arXiv, March 20, 2018. http://arxiv.org/abs/1802.09984.

Graph Query Language GQL. "GQL Standard." Last modified December 2, 2022. https://www.gqlstandards.org/home

Longley, John. "Goals of formal semantics." The University of Edinburgh School of Informatics. Accessed January 28, 2023, https://www.inf.ed.ac.uk/teaching/courses/fpls/note2.pdf

Memgraph. "Neo4j vs Memgraph." Accessed January 28, 2023. https://memgraph.com/memgraph-for-neo4j-developers

Neo4j. "Cypher Query Language." Accessed January 28, 2023. https://neo4j.com/developer/cypher/

Redis. "RedisGraph: A Graph database built on Redis." Accessed January 28, 2023. https://redis.io/docs/stack/graph/

Appendix A

The code can be seen at this GitHub link:
https://github.com/sysemenova/cypher_formal_semantics