Cypher: An Evolving Query Language for Property Graphs

Nadime Francis*
Université Paris-Est

Alastair Green
Neo4j

Paolo Guagliardo
University of Edinburgh

Leonid Libkin
University of Edinburgh

Tobias Lindaaker
Neo4j

Victor Marsault
University of Edinburgh

Stefan Plantikow
Neo4j

Mats Rydberg
Neo4j

Petra Selmer
Neo4j

Andrés Taylor
Neo4j

ABSTRACT

The Cypher property graph query language is an evolving language, originally designed and implemented as part of the Neo4j graph database, and it is currently used by several commercial database products and researchers. We describe Cypher 9, which is the first version of the language governed by the openCypher Implementers Group. We first introduce the language by example, and describe its uses in industry. We then provide a formal semantic definition of the core read-query features of Cypher, including its variant of the property graph data model, and its "ASCII Art" graph pattern matching mechanism for expressing subgraphs of interest to an application. We compare the features of Cypher to other property graph query languages, and describe extensions, at an advanced stage of development, which will form part of Cypher 10, turning the language into a compositional language which supports graph projections and multiple named graphs.

ACM Reference Format:
Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3190657

1 INTRODUCTION

In the last decade, property graph databases [34] such as Neo4j, JanusGraph and Sparksee have become more widespread in industry and academia. They have been used in multiple domains, such as master data and knowledge management, recommendation engines, fraud detection, IT operations and network management, authorization and access control [52], bioinformatics [39], social networks [17], software system analysis [25], and in investigative journalism [11]. Using graph databases to manage graph-structured data confers many benefits such as explicit support for modeling graph data, native indexing and storage for fast graph traversal operations, built-in support for graph algorithms (e.g., Page Rank, subgraph matching and so on), and the provision of graph languages, allowing users to express complex pattern-matching operations.

In this paper we describe Cypher, a well-established language for querying and updating property graph databases, which began life in the Neo4j product, but has now been implemented commercially in other products such as SAP HANA Graph, Redis Graph, Agens Graph (over PostgreSQL) and Memgraph. Neo4j [52] is one of the most popular property graph databases[1] that stores graphs natively on disk and provides a framework for traversing graphs and executing graph operations. The language therefore is used in hundreds of production applications across many industry vertical domains. Cypher is also used in several research projects (e.g., Ingraph [41], Gradoop [29], and Cytosm [55]) as well as in recent or incubating open-source projects, such as Cypher for Apache Spark and Cypher over Gremlin.

Since 2015 the openCypher project[2] has sought to enable the use of Cypher as a standardized language capable of multiple independent implementations, and to provide a framework for cross-implementer collaborative evolution of new language features. The goal is that Cypher will mature into an industry standard language for property graph querying, playing a complementary role to that of the SQL standard for relational data querying. Here we present Cypher 9 [47], the first version of the language governed by openCypher. We give an introduction to the language, describe its uses in industry, provide a formal definition of its data model and the semantics of its queries and clauses, and then describe current work that will lead to Cypher 10, a compositional language supporting graph projections and multiple named graphs.

The data model of Neo4j that is used by Cypher is that of *property graphs*. It is the most popular graph data model in industry, and is becoming increasingly prevalent in academia [38]. The model comprises *nodes*, representing entities (such as people, bank accounts, departments and so on), and *relationships* (synonymous with *edges*), representing the connections or relationships between the entities. In the graph model, the relationships are as important as the entities themselves. Moreover, any number of attributes (henceforth termed *properties*), in the form of key-value pairs, may be associated

*Affiliated with the School of Informatics at the University of Edinburgh during the time of contributing to this work.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06.
https://doi.org/10.1145/3183713.3190657

[1] https://db-engines.com/en/ranking/graph+dbms
[2] https://www.opencypher.org

with the nodes and relationships. This allows for the modeling and querying of complex data.

The language comes with a fully formalized semantics of its core constructs. The need for it stems from the fact that Cypher, in addition to being implemented in an industrial product with a significant customer base, has been picked up by others, and several implementations of it exist. Given the lack of a standard for the language (which can take many years to complete, as it did for SQL), it has become pressing to agree on the formal data model and the meaning of the main constructs. A formal semantics has other advantages too; for example, it allows one to reason about the equivalence of queries, and prove correctness of existing or discover new optimizations. The need of the formal semantics has long been accepted in the field of programming languages [43] and for several common languages their semantics has been fully worked out [1, 22, 42, 49]. Recently similar efforts have been made for the core SQL constructs [12, 13, 20, 57] with the goal of proving correctness of SQL optimizations and understanding the expressiveness of its features. The existence of the formal semantics of Cypher makes it possible for different implementations to agree on its core features, and paves a way to a reference implementation against which others will be compared. We also note that providing semantics for an existing real-life language like Cypher that accounts for all of its idiosyncrasies is much harder than for theoretical calculi underlying main features of languages, as has been witnessed by previous work on SQL [20] and on many programming languages.

Organization. We provide a high-level overview of Cypher features in Section 2. In Section 3 we illustrate the semantics of the major clauses by means of a step-by-step analysis of an example query, and give further examples of queries used in industrial applications. The formal specification of the core of Cypher is given in Section 4. The historical context of Cypher is presented in Section 5, followed by a description of current developments in Cypher, such as query composition and support for multiple graphs, in Section 6. In Section 7 we cover related work on other graph data models and graph query languages. Future work and conclusions are given in Sections 8 and Section 9.

2 THE CYPHER LANGUAGE

Cypher is a declarative query language for property graphs. Cypher provides capabilities for both querying and modifying data, as well as specifying schema definitions.

Linear queries. A Cypher query takes as input a property graph and outputs a table. These tables can be thought of as providing bindings for parameters that witness some patterns in a graph, with some additional processing done on them.

Cypher structures queries linearly. This allows users to think of query processing as starting from the beginning of the query text and then progressing linearly to the end. Each clause in a query is a function that takes a table and outputs a table that can both expand the number of fields and add new tuples. The whole query is then the composition of these functions. This linear order of clauses is understood purely declaratively – implementations are free to re-order the execution of clauses if this does not change the semantics of the query. Thus, rather than declaring the projection at the beginning of the query like SQL does with **SELECT**, in Cypher the projection goes at the end of the query as **RETURN**.

The linear flow of queries in Cypher extends to query composition. By using **WITH**, the query continues with the projected table from the query part before **WITH** as the driving table for the query part after **WITH**. The **WITH** clause allows the same projections as **RETURN**, including aggregations. It also supports filtering based on the projected fields, as we shall see in Section 3.

In addition to this linear way of composing queries, Cypher also supports nested subqueries such as **UNION** queries.

Pattern matching. The central concept in Cypher queries is pattern matching. Patterns in Cypher are expressed in a visual form as "ASCII art", such as `(a)-[r]->(b)`. The **MATCH** clause in Cypher uses such a pattern and introduces new rows (synonymous with *records*) with bindings of the matched instances of the pattern in the queried graph.

Cypher's functionality was influenced by XPath [30] and SPARQL [58], and its patterns express a restricted form of regular path queries: the concatenation and disjunction of single relationship types, as well as variable length paths (essentially, transitive closure). Cypher also supports matching and returning paths as values.

Data modification. Cypher supports a rich update language for modifying the graph. Updating clauses re-use the visual graph-pattern language and provide the same simple, top-down semantic model as the rest of Cypher. The basic clauses for updates include **CREATE** for creating new nodes and relationships, **DELETE** for removing entities, and **SET** for updating properties. Additionally, Cypher provides a clause called **MERGE** which tries to match the given pattern, and creates the pattern if no match was found. An implementation of Cypher can use database synchronization primitives such as locking to ensure that patterns matched by **MERGE** are unique within the database.

Pragmatic. Cypher is intentionally similar to SQL in order to help users transition between the two languages. It follows the same clause syntax structure and implements the established semantics for many functions. Cypher has built-in support for query parameters, making it easy to eliminate the problems of query injection. The choice of defaults aligns with common usage. The syntax for grouping and aggregation is simple, and the expression language includes powerful features such as list slicing and list comprehensions, existential subqueries, and working with paths.

Neo4j implementation. Query execution in Neo4j follows a conventional model, outlined by the Volcano Optimizer Generator [19]. The query planning in Neo4j is based on the IDP algorithm [44, 54], using a cost model described in [21]. The final query compilation uses either a simple tuple-at-a-time iterator-based execution model, or compiles the query to Java bytecode with a push-based execution model based on [46].

An execution plan for a Cypher query in Neo4j contains largely the same operators as in relational database engines and an additional operator called *Expand*. Semantically *Expand* is very similar to a relational join. It finds pairs of nodes that are connected through an edge. In terms of implementation it utilizes the fact that the data representation of Neo4j contains direct references from each node via its edges to the related nodes. This means that *Expand* never

needs to read any unnecessary data, or proceed via an indirection such as an index in order to find related nodes.

3 CYPHER BY EXAMPLE

Figure 1 shows a data graph G consisting of researchers, students and publications. For each researcher we show the students they supervise, and the publications they have authored, and for each publication, we show which other publications it cites.

The following Cypher query returns the name of each researcher in G, the number of students they currently supervise, and the number of times a publication they have authored has been cited – both directly and indirectly – by other publications.

```
1 MATCH (r:Researcher)
2 OPTIONAL MATCH (r)-[:SUPERVISES]->(s:Student)
3 WITH r, count(s) AS studentsSupervised
4 MATCH (r)-[:AUTHORS]->(p1:Publication)
5 OPTIONAL MATCH (p1)<-[:CITES*]-(p2:Publication)
6 RETURN r.name, studentsSupervised,
7        count(DISTINCT p2) AS citedCount
```

The pattern given in the first **MATCH** clause in line 1 matches all researchers; i.e., nodes with the label `Researcher`. This produces three bindings for the variable `r`, namely n_1, n_6, and n_{10} represented as three rows in a table with a single attribute r, that correspond to the researchers *Nils*, *Elin* and *Thor*.

The **MATCH** clause has an optional variant: **OPTIONAL MATCH**, which is analogous to the outer join construct in SQL. This clause produces rows for all matches in the same way that **MATCH** does, providing the *entire* pattern is found in the data graph. However, in cases where no data matching the entire pattern is found, a single row will be produced in which the bindings for all variables introduced in **OPTIONAL MATCH** will be set to `null`.

r	s
n_1	null
n_6	n_7
n_6	n_8
n_{10}	n_7

(a)

r	studentsSupervised
n_1	0
n_6	2
n_{10}	1

(b)

Figure 2: Variable bindings

The **OPTIONAL MATCH** clause in line 2 matches all the students supervised by each researcher. This yields a binding of the newly-introduced variable `s` for each value to which `r` was bound by the **MATCH** clause in line 1, producing the table in Figure 2a. When `r` is bound to n_1 (*Nils*, who does not supervise any students), the corresponding binding for `s` is `null`. For the researcher *Elin* (n_6), who supervises two students, both n_7 and n_8 are bound.

The **WITH** clause in line 3 is used both to project a subset of the variables currently in scope and their bindings to the query part succeeding **WITH**, and to compute an aggregation. The **WITH** clause has two expressions, the second of which is an aggregation, functioning in a very similar way to SQL. The first expression, `r`, is a non-aggregating expression and therefore acts as an implicit *grouping key* for the aggregating function **count**`(s)`. Taking the results shown in the above table, we count all the non-`null` values of `s` for each unique binding of `r`, aliasing it as `studentsSupervised`. The **WITH** clause will project all the bindings produced for `r` and `studentsSupervised`, as shown in Figure 2b. We note that the variable `s` is no longer in scope after line 3, as it was not projected by **WITH**, and may no longer be used in the remainder of the query.

This table now acts as the driving table for the **MATCH** clause in line 4. Here, we define a pattern which matches all the publications authored by the researchers. The researcher *Thor* has not authored any publications, which means n_{10} will not be matched. As a result, n_{10} is not produced as a binding for `r`, and therefore no row containing these bindings will be added to the intermediate result table. The variable bindings produced by line 4 include all researchers who have authored at least one publication, the number of students they supervise, and the publication they authored:

r	studentsSupervised	p1
n_1	0	n_2
n_6	2	n_5
n_6	2	n_9

The **OPTIONAL MATCH** clause in line 5 matches, for each publication authored by one of the researchers in G, all the publications which cite it, both directly and indirectly. This is achieved through the use of a pattern containing the *variable-length* relationship `CITES*`, indicating that one or more `CITES` relationships must be traversed. The results produced are:

r	studentsSupervised	p1	p2	
n_1	0	n_2	n_4	
n_1	0	n_2	n_9	†
n_1	0	n_2	n_5	
n_1	0	n_2	n_9	†
n_6	2	n_5	n_9	
n_6	2	n_9	null	

When `p1` is bound to n_9, which is not cited by any publication, the corresponding binding for `p2` is `null`. In addition, we note that there are two identical rows, indicated with †. The existence of these duplicate rows is due to the variable-length relationship pattern: n_9 is reachable from n_2 through the intermediate nodes n_5 and n_4.

RETURN is the last clause in the query (lines 6 and 7) and, much like the **WITH** clause in line 3, computes and projects expressions. The value of the `name` property of each researcher is projected, along with the value of `studentsSupervised`. The aggregation expression, in contrast to the one in line 3, counts the distinct values of `p2` (excluding `null` values) and aliases the results as `citedCount`, denoting the number of publications citing a publication authored by the researcher. The table consisting of the rows containing the result of the expressions is projected by **RETURN**:

r.name	studentsSupervised	citedCount
Nils	0	3
Elin	2	1

Examples from industry. Bioinformatics [3, 61] and pharmaceuticals [45] are major areas where graph storage and analytics are

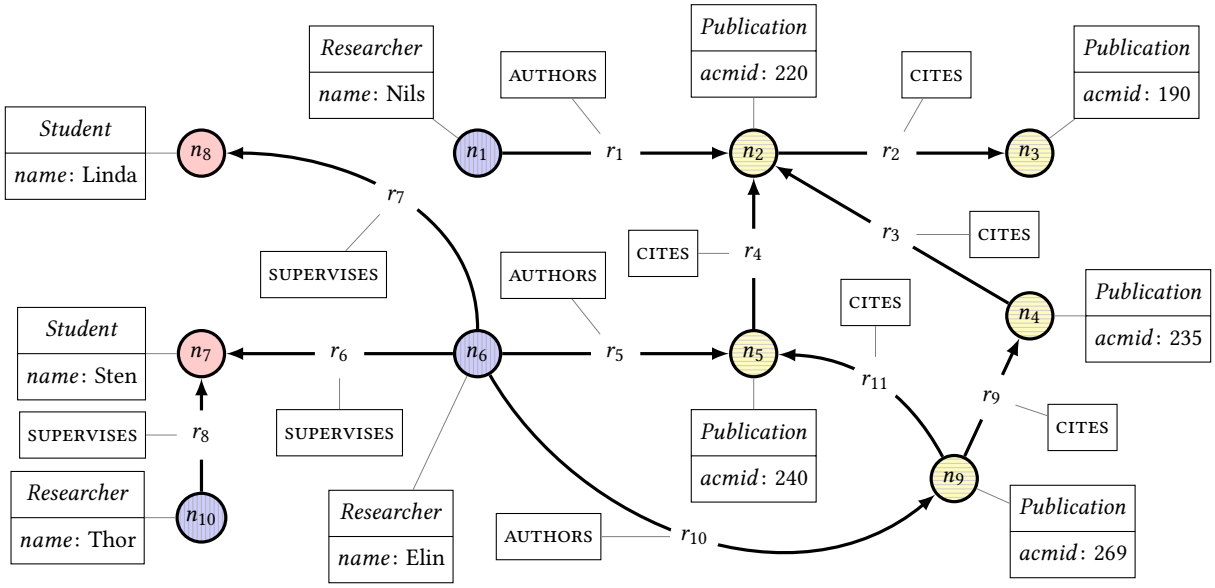

Figure 1: Example data graph showing supervision and citation data for researchers, students and publications

widely used. There is a strong overlap between graph processing and machine learning [27]. Several specific application domains have played a large role in the expansion of property graphs and the use of the Cypher language: fraud detection, knowledge management, network and IT operations, real-time recommendation engines, master data management, social networks, and identity and access management.

A real-world query from the network management domain is shown below. In a data center, entities such as services, firewalls, servers, routers and network switches are modeled as nodes, with relationships representing the dependencies between them. The query returns the component that is depended upon – both directly and indirectly – by the largest number of entities.

```
MATCH (svc:Service)<-[:DEPENDS_ON*]-(dep:Service)
RETURN svc, count(DISTINCT dep) AS dependents
ORDER BY dependents DESC
LIMIT 1
```

Our second example shows a query in the domain of fraud detection. It returns details regarding a potential fraud ring, in which distinct account holders share personal information, such as social security number (SSN), telephone number and address.

```
MATCH (accHolder:AccountHolder)-[:HAS]->(pInfo)
WHERE pInfo:SSN OR pInfo:PhoneNumber
                OR pInfo:Address
WITH pInfo,
    collect(accHolder.uniqueId) AS accountHolders,
    count(*) AS fraudRingCount
WHERE fraudRing > 1
RETURN accountHolders,
    labels(pInfo) AS personalInformation,
    fraudRingCount
```

The `collect` function returns a list containing the values returned by the expression, and the `labels` function returns a list containing all the labels of a node. Thus, in the query above, these functions will return, respectively, a list containing the unique identifiers of all the account holders, and a list containing all the labels of the nodes bound to the variable `pInfo`.

4 FORMAL SPECIFICATION

We now provide a formal specification of the core of Cypher. Formal specifications of languages have multiple advantages over documentation written in natural language. They can be used to reason about the language and prove optimizations correct, they can lead to a reference implementation that could be used to verify whether a particular implementation of the language adheres to its specification, and they leave much less room for ambiguity compared to natural language descriptions. While common in the programming language community, formal specifications of query languages have recently appeared in the database literature [12, 13, 20, 57].

The key elements of Cypher are as follows:

- data model, that includes *values*, *graphs*, and *tables*;
- query language, that includes *expressions*, *patterns*, *clauses*, and *queries*.

Values could be simple or composite, such as lists and maps; we have already seen property graphs, and tables that are the outputs of queries. Expressions denote values; patterns occur in `MATCH` clauses; and queries are sequences of clauses. Each clause denotes a function from tables to tables.

To provide a formal semantics of Cypher, we need to define one relation and two functions:

- The *pattern matching relation* checks if a path p in a graph G satisfies a pattern π, under an assignment u of values to the free variables of the pattern. This is written as $(p, G, u) \models \pi$.

Table 1: Summary of notational conventions

Concept	Notation	Set notation
Property keys	k	$\mathcal{K}$
Node identifiers	n	$\mathcal{N}$
Relationship identifiers	r	$\mathcal{R}$
Node labels	ℓ	$\mathcal{L}$
Relationship types	t	$\mathcal{T}$
Names	a	$\mathcal{A}$
Base functions	f	$\mathcal{F}$
Values	v	$\mathcal{V}$
Node patterns	χ	–
Relationship patterns	ρ	–
Path patterns	π	–

- The *semantics of expressions* associates an expression expr, a graph G and an assignment u with a value $[\![\text{expr}]\!]_{G,u}$.
- The *semantics of queries* (resp., *clauses*) associates a query Q (resp., clause C) and a graph G with a function $[\![Q]\!]_G$ (resp., $[\![C]\!]_G$) that takes a table and returns a table (perhaps with more rows or with wider rows).

Note that the semantics of a query Q is a *function*; thus it should not be confused with the *output* of Q. The evaluation of a query starts with the table containing one empty tuple, which is then progressively changed by applying functions that provide the semantics of Q's clauses. The composition of such functions, i.e., the semantics of Q, is a function again, which defines the output as

$$\text{output}(Q, G) \;=\; [\![Q]\!]_G(T_{()})$$

where $T_{()}$ is the table containing the single empty tuple ().

With this basic understanding of the data model and the semantics of the language, we now explain it in detail. Throughout the description of the semantics, we shall use the notational conventions in Table 1 (they will be explained in the following sections; they are summarized here for a convenient reference).

4.1 Data Model: values, graphs, tables

Values. We consider three disjoint sets $\mathcal{K}$ of *property keys*, $\mathcal{N}$ of *node identifiers* and $\mathcal{R}$ of *relationship identifiers* (ids for short). These sets are all assumed to be countably infinite (so we never run out of keys and ids). For this presentation of the model, we assume two base types: the integers $\mathbb{Z}$, and the type of finite strings over a finite alphabet Σ (this does not really affect the semantics of queries; these two types are chosen purely for illustration purposes).

The set $\mathcal{V}$ of values is inductively defined as follows:

- Identifiers (i.e., elements of $\mathcal{N}$ and $\mathcal{R}$) are values;
- Base types (elements of $\mathbb{Z}$ and Σ^*) are values;
- `true`, `false` and `null` are values;
- list() is a value (empty list), and if $v_1, \ldots, v_m$ are values, for $m > 0$, then list$(v_1, \ldots, v_m)$ is a value.
- map() is a value (empty map), and if $k_1, \ldots, k_m$ are distinct property keys and $v_1, \ldots, v_m$ are values, for $m > 0$, then map$((k_1, v_1), \ldots, (k_m, v_m))$ is a value.
- If n is a node identifier, then path(n) is a value. If $n_1, \ldots, n_m$ are node ids and $r_1, \ldots, r_{m-1}$ are relationship ids, for $m > 1$, then path$(n_1, r_1, n_2, \ldots, n_{m-1}, r_{m-1}, n_m)$ is a value. We shall use shorthands n and $n_1 r_1 n_2 \ldots n_{m-1} r_{m-1} n_m$.

In the Cypher syntax, lists are $[v_1, \ldots, v_m]$ and maps are $\{k_1 : v_1, \ldots, k_m : v_m\}$; we use explicit notation for them to make clear the distinction between the syntax and the semantics of values.

We use the symbol "$\cdot$" to denote concatenation of paths, which is possible only if the first path ends in a node where the second starts, i.e., if $p_1 = n_1 r_1 \cdots r_{j-1} n_j$ and $p_2 = n_j r_j \cdots r_{m-1} n_m$ then $p_1 \cdot p_2$ is $n_1 r_1 n_2 \cdots n_{m-1} r_{m-1} n_m$.

Every real-life query language will have a number of functions defined on its values, e.g., concatenation of strings and arithmetic operations on numbers. To model this, we assume a finite set $\mathcal{F}$ of predefined functions that can be applied to values (and produce new values). The semantics is parameterized by this set, which can be extended whenever new types and/or basic functions are added to the language.

Property graphs. Let $\mathcal{L}$ and $\mathcal{T}$ be countable sets of node labels and relationship types, respectively. A property graph is a tuple $G = \langle N, R, \text{src}, \text{tgt}, \iota, \lambda, \tau \rangle$ where:

- N is a finite subset of $\mathcal{N}$, whose elements are referred to as the *nodes* of G.
- R is a finite subset of $\mathcal{R}$, whose elements are referred to as the *relationships* of G.
- $\text{src} : R \to N$ is a function that maps each relationship to its *source* node.
- $\text{tgt} : R \to N$ is a function that maps each relationship to its *target* node.
- $\iota : (N \cup R) \times \mathcal{K} \to \mathcal{V}$ is a finite partial function that maps a (node or relationship) identifier and a property key to a value.
- $\lambda : N \to 2^{\mathcal{L}}$ is a function that maps each node id to a finite (possibly empty) set of labels.
- $\tau : R \to \mathcal{T}$ is a function that maps each relationship identifier to a relationship type.

Example 4.1. We now refer to the property graph in Figure 1 and show how, for a sample of its nodes and relationships, it is formally represented in this model as a graph $G = (N, R, \text{src}, \text{tgt}, \iota, \lambda, \tau)$.

- $N = \{n_1, \ldots, n_{10}\}$;
- $R = \{r_1, \ldots, r_{11}\}$;
- $\text{src} = \left\{ \begin{array}{llll} r_1 \mapsto n_1\,, & r_4 \mapsto n_5\,, & r_7 \mapsto n_6\,, & r_{10} \mapsto n_6 \\ r_2 \mapsto n_2\,, & r_5 \mapsto n_6\,, & r_8 \mapsto n_{10}\,, & r_{11} \mapsto n_9 \\ r_3 \mapsto n_4\,, & r_6 \mapsto n_6\,, & r_9 \mapsto n_9 & \end{array} \right\}$;
- $\text{tgt} = \left\{ \begin{array}{llll} r_1 \mapsto n_2\,, & r_4 \mapsto n_2\,, & r_7 \mapsto n_8\,, & r_{10} \mapsto n_9 \\ r_2 \mapsto n_3\,, & r_5 \mapsto n_5\,, & r_8 \mapsto n_7\,, & r_{11} \mapsto n_5 \\ r_3 \mapsto n_2\,, & r_6 \mapsto n_7\,, & r_9 \mapsto n_4 & \end{array} \right\}$;
- $\iota(n_1, \text{name}) = \text{Nils}$, $\iota(n_2, \text{acmid}) = 220$, $\iota(n_3, \text{acmid}) = 190$, …, $\iota(n_{10}, \text{name}) = \text{Thor}$;
- $\lambda(n_1) = \lambda(n_6) = \lambda(n_{10}) = \{\text{Student}\}$, $\lambda(n_2) = \lambda(n_3) = \lambda(n_4) = \lambda(n_5) = \lambda(n_9) = \{\text{Publication}\}$, $\lambda(n_7) = \lambda(n_8) = \{\text{Researcher}\}$;
- $\tau(r) = \begin{cases} \text{AUTHORS} & \text{for } r \in \{r_1, r_5, r_{10}\}\,, \\ \text{SUPERVISES} & \text{for } r \in \{r_6, r_7, r_8\}\,, \\ \text{CITES} & \text{for } r \in \{r_2, r_3, r_4, r_9, r_{11}\}\,. \end{cases}$

Tables. Let $\mathcal{A}$ be a countable set of names. A *record* is a partial function from names to values, conventionally denoted as a tuple with named fields $u = (a_1 : v_1, \ldots, a_n : v_n)$ where $a_1, \ldots, a_n$ are

distinct names, and $v_1, \ldots, v_n$ are values. The order in which the fields appear is only for notation purposes. We refer to $\mathrm{dom}(u)$, i.e., the domain of u, as the set $\{a_1, \ldots, a_m\}$ of names used in u. Two records u and u' are *uniform* if $\mathrm{dom}(u) = \mathrm{dom}(u')$.

If $u = (a_1 : v_1, \ldots, a_n : v_n)$ and $u' = (a'_1 : v'_1, \ldots, a_m : v'_m)$ are two records, then (u, u') denotes the record $(a_1 : v_1, \ldots, a_n : v_n, a'_1 : v'_1, \ldots, a'_m : v'_m)$, assuming that all a_i, a'_j for $i \leq n, j \leq m$ are distinct. If $A = \{a_1, \ldots, a_n\}$ is a set of names v is a value, then $(A : v)$ denotes the record $(a_1 : v, \ldots, a_n : v)$. We use $()$ to denote the empty record, i.e., the partial function from names to values whose domain is empty.

If A is a set of names, then a *table* with fields A is a bag, or multiset, of records u such that $\mathrm{dom}(u) = A$. A table with no fields is just a bag of copies of the empty record. In most cases, the set of fields of tables will be clear from the context, and will not be explicitly stated. Given two tables T and T', we use $T \uplus T'$ to denote their *bag union*, in which the multiplicity of each record is the sum of their multiplicities in T and T'. If $B = \{b_1, \ldots, b_n\}$ is a bag, and $T_{b_1}, \ldots, T_{b_n}$ are tables, then $\biguplus_{b \in B} T_b$ stands for $T_{b_1} \uplus \ldots \uplus T_{b_n}$. Finally, we use $\varepsilon(T)$ to denote the result of duplicate elimination on T, i.e., each tuple of T is present just once in $\varepsilon(T)$.

4.2 Patterns and Pattern Matching

Syntax of patterns. It is important to remember that the Cypher grammar is defined by mutual recursion of expressions, patterns, clauses, and queries. Here, the description of patterns will make a reference to expressions, which we will cover later on; all we need to know for now is that these will denote values.

The Cypher syntax of patterns is given in Figure 3, where the highlighted symbols denote tokens of the language. Instead of the actual Cypher syntax, here we use an abstract mathematical notation that lends itself more naturally to a formal treatment.

A node pattern χ is a triple (a, L, P) where:

- $a \in \mathcal{A} \cup \{\mathsf{nil}\}$ is an optional name;
- $L \subset \mathcal{L}$ is a possibly empty finite set of node labels;
- P is a possibly empty finite partial map from $\mathcal{K}$ to expressions.

For example, the following node pattern in Cypher syntax:

```
(x:Person:Male {name: expr1, age: expr2})
```

is represented as $(x, \{\text{Person}, \text{Male}\}, \{\text{name} \mapsto e_1, \text{age} \mapsto e_2\})$, where e_1 and e_2 are the representations of expressions expr_1 and expr_2, respectively. The simplest node pattern `()` is represented by $(\mathsf{nil}, \varnothing, \varnothing)$.

A relationship pattern ρ is a tuple (d, a, T, P, I) where:

- $d \in \{\rightarrow, \leftarrow, \leftrightarrow\}$ specifies the *direction* of the pattern: left-to-right ($\rightarrow$), right-to-left ($\leftarrow$), or undirected ($\leftrightarrow$);
- $a \in \mathcal{A} \cup \{\mathsf{nil}\}$ is an optional name,
- $T \subset \mathcal{T}$ is a possibly empty finite set of relationship types;
- P is a possibly empty finite partial map from $\mathcal{K}$ to expressions;
- I is either nil or (m, n) with $m, n \in \mathbb{N} \cup \{\mathsf{nil}\}$.

For instance, the following relationship patterns in Cypher syntax:

```
-[:KNOWS*1 {since: 1985}]-
-[:KNOWS*1..1 {since: 1985}]-
```

are both represented as $(\leftrightarrow, \mathsf{nil}, \{\text{KNOWS}\}, \{\text{since} \mapsto 1985\}, (1, 1))$. However, the following relationship pattern:

```
-[:KNOWS {since: 1985}]-
```

will be represented as $(\leftrightarrow, \mathsf{nil}, \{\text{KNOWS}\}, \{\text{since} \mapsto 1985\}, \mathsf{nil})$. As highlighted by this example, I is nil iff the optional grammar token len does not appear in the pattern syntax (see Figure 3). Otherwise, I is equal to (nil, nil) if len derives to $*$, and I is equal to (d, d), (d_1, nil), (nil, d_2), (d_1, d_2) if other derivations rules are applied, respectively.

In general, I defines the range of the relationship pattern. The range is $[m, n]$ if $I = (m, n)$ where nil is replaced by 1 and ∞ in the place of the lower and upper bounds. The range is $[1, 1]$ if $I = \mathsf{nil}$. A relationship pattern is said *rigid* if its range [m,n] satisfies $m = n \in \mathbb{N}$.

A path pattern is an alternating sequence of the form

$$\chi_1 \; \rho_1 \; \chi_2 \; \cdots \; \rho_{n-1} \; \chi_n$$

where each χ_i is a node pattern and each ρ_i is a relationship pattern. A path pattern π can be optionally given a name a, written as π/a; we then refer to a *named pattern*. A path pattern is *rigid* if all relationship patterns in it are rigid, and *variable length* otherwise.

We shall now define the satisfaction relation for path patterns w.r.t. a property graph $G = (N, R, \mathrm{src}, \mathrm{tgt}, \iota, \lambda, \tau)$, a path p with node ids from N and relationship ids from R, and an assignment u.

We consider rigid patterns first as a special case, because they – unlike variable length patterns – uniquely define both the length and the possible variable bindings of the paths satisfying them. The satisfaction of variable length patterns will then be defined in terms of a set of rigid patterns.

Satisfaction of rigid patterns. As a precondition for a path p to satisfy any pattern, it is necessary that *all relationships in p are distinct*. Then, the definition is inductive, with the base case given by node patterns (which are rigid path patterns). Let χ be a node pattern (a, L, P); then $(n, G, u) \models \chi$ if all of the following hold:

- either a is nil or $u(a) = n$;
- $L \subseteq \lambda(n)$;
- $[\![\iota(n, k) = P(k)]\!]_{G,u} = \texttt{true}$ for each k s.t. $P(k)$ is defined.

Example 4.2. Consider the property graph G in Figure 4 and the node patterns $\chi_1 = (x, \{\text{Teacher}\}, \varnothing)$ and $\chi_2 = (y, \varnothing, \varnothing)$. Then,

$(n_1, G, u) \models \chi_1$ if u is an assignment that maps x to n_1,
$(n_2, G, u) \not\models \chi_1$ for any assignment u,
$(n_3, G, u) \models \chi_1$ if u is an assignment that maps x to n_3,
$(n_4, G, u) \models \chi_1$ if u is an assignment that maps x to n_4.

For $i = 1, \ldots, 4$ we have that $(n_i, G, u_i) \models \chi_2$ whenever u_i is an assignment that maps y to n_i. □

For the inductive case, let χ be a node pattern, let π be a rigid path pattern, and let ρ be the relationship pattern (d, a, T, P, I). First we assume that $I \neq \mathsf{nil}$; since ρ is rigid, $I = (m, m)$ with $m \in \mathbb{N}$. For $m = 0$, we have that $(n \cdot p, G, u) \models \chi\rho\pi$ if

(a) either a is nil or $u(a) = \mathrm{list}()$; and
(b) $(n, G, u) \models \chi$ and $(p, G, u) \models \pi$.

For $m \geq 1$, we have that $(n_1 r_1 n_2 \cdots r_m n_{m+1} \cdot p, G, u) \models \chi\rho\pi$ if all of the following hold:

(a′) either a is nil or $u(a) = \mathrm{list}(r_1, \ldots, r_m)$;
(b′) $(n_1, G, u) \models \chi$ and $(p, G, u) \models \pi$;

and, for every $i \in \{1, \ldots, m\}$, all of the following hold:

(c′) $\tau(r_i) \in T$;

pattern ::= pattern° | a = pattern°

pattern° ::= node_pattern | node_pattern rel_pattern pattern°

node_pattern ::= (a? label_list? map?)

rel_pattern ::= -[a? type_list? len? map?]->
| <-[a? type_list? len? map?]-
| -[a? type_list? len? map?]-

label_list ::= :ℓ | :ℓ label_list

map ::= { prop_list }

prop_list ::= k : expr | k : expr , prop_list

type_list ::= :t | type_list | t

len ::= * | *d | *d1 . . | *. . d2 | *d1 . . d2

d, d1, d2 ∈ ℕ

Figure 3: Syntax of Cypher patterns

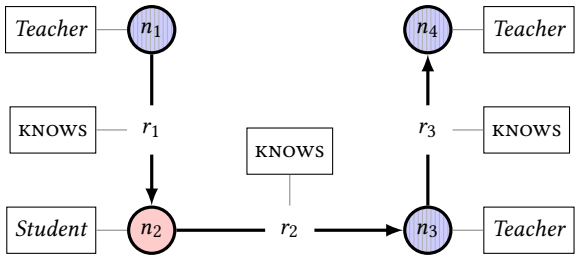

Figure 4: Property graph with students and teachers

(d′) $[\![\iota(r_i, k) = P(k)]\!]_{G,u}$ = **true** for every k s.t. $P(k)$ is defined;

(e′) $(\mathrm{src}(r_i), \mathrm{tgt}(r_i)) \in \begin{cases} \{(n_i, n_{i+1}), (n_{i+1}, n_i)\} & \text{if } d \text{ is } \leftrightarrow \ , \\ \{(n_i, n_{i+1})\} & \text{if } d \text{ is } \rightarrow \ , \\ \{(n_{i+1}, n_i)\} & \text{if } d \text{ is } \leftarrow \ . \end{cases}$

The case when $I = \mathrm{nil}$ is treated as if $I = (1, 1)$, except that item (a′) above is replaced by: (a″) either a is nil or $u(a) = r_1$.

Example 4.3. Consider again the property graph G in Figure 4 and the following rigid pattern π in Cypher syntax:

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

In our mathematical representation this amounts to:

$$\underbrace{(x, \{\text{Teacher}\}, \varnothing)}_{\chi_1}, \underbrace{(\rightarrow, \mathrm{nil}, \{\text{KNOWS}\}, \varnothing, (2, 2))}_{\rho}, \underbrace{(y, \varnothing, \varnothing)}_{\chi_2}$$

where χ_1 and χ_2 are the node patterns we have seen in Example 4.2. Now, let $u = \{x \mapsto n_1, y \mapsto n_3\}$; from that example we know that $(n_1, G, u) \models \chi_1$ and $(n_3, G, u) \models \chi_2$. Then, following the definition of satisfaction given above, one can easily see that $(p, G, u) \models \pi$, where $p = n_1 r_1 n_2 r_2 n_3$ and $\pi = \chi_1 \rho \chi_2$.

Observe that if there is another assignment u' s.t. $(p, G, u') \models \pi$, then u' maps x to n_1 and y to n_3. This is the intuitive reason why rigid patterns are of interest: given a path and a rigid pattern, there exists at most one possible assignment of the *free variables* (which we shall formally define shortly) of the pattern w.r.t. which the path satisfies the pattern. We will see that for variable length patterns this is no longer the case. □

For named rigid patterns, we have that $(p, G, u) \models \pi/a$ if $u(a) = p$ and $(p, G, u) \models \pi$.

Satisfaction of variable length patterns. Informally, a variable length pattern is a compact representation for a possibly infinite set of rigid patterns; e.g., a pattern of length *at least* 1 will represent patterns of length 1, patterns of length 2, and so on.

To make this idea precise, let $\rho = (d, a, T, P, I)$ be a variable length relationship pattern, and $\rho' = (d, a, T, P, (m', m'))$ be a rigid relationship pattern. We say that ρ *subsumes* ρ', and write $\rho \sqsupset \rho'$, if m' belongs to the range $[m, n]$ defined by I. If ρ is rigid, then it only subsumes itself. This subsumption relation is easily extended to path patterns. Given a variable length pattern $\pi = \chi_1 \rho_1 \chi_2 \cdots \rho_{k-1} \chi_k$ and a rigid pattern $\pi' = \chi_1 \rho'_1 \chi_2 \cdots \rho'_{k-1} \chi_k$, we say that π subsumes π' (written $\pi \sqsupset \pi'$) if $\rho_i \sqsupset \rho'_i$ for every $i \in \{1, \ldots, k-1\}$.

Then, we define the *rigid extension* of π as

$$\mathrm{rigid}(\pi) = \{\, \pi' \mid \pi' \text{ is rigid and } \pi \sqsupset \pi' \,\} \ ,$$

that is, the (possibly infinite) set of all rigid patterns subsumed by π. For a named pattern, $\mathrm{rigid}(\pi/a) = \{\pi'/a \mid \pi' \in \mathrm{rigid}(\pi)\}$. Finally, $(p, G, u) \models \pi$ if $(p, G, u) \models \pi'$ for some $\pi' \in \mathrm{rigid}(\pi)$, and similarly for named patterns.

Example 4.4. Consider the following variable length pattern π:

```
(x:Teacher) -[:KNOWS*1..2]-> (z)
            -[:KNOWS*1..2]-> (y:Teacher)
```

That is, π is the pattern $\chi_1 \rho \chi_2 \rho \chi_3$ with

$$\chi_1 = (x, \{\text{Teacher}\}, \varnothing) \ , \quad \chi_3 = (y, \{\text{Teacher}\}, \varnothing) \ ,$$
$$\chi_2 = (z, \varnothing, \varnothing) \ , \quad \rho = (\rightarrow, \mathrm{nil}, \{\text{KNOWS}\}, \varnothing, (1, 2)) \ .$$

Then, $\mathrm{rigid}(\pi)$ is the set

$$\Big\{ \underbrace{\chi_1 \rho_1 \chi_2 \rho_1 \chi_3}_{\pi_1}, \underbrace{\chi_1 \rho_1 \chi_2 \rho_2 \chi_3}_{\pi_2}, \underbrace{\chi_1 \rho_2 \chi_1 \rho_1 \chi_3}_{\pi_3}, \underbrace{\chi_1 \rho_2 \chi_2 \rho_2 \chi_3}_{\pi_4} \Big\}$$

where

$$\rho_1 = (\rightarrow, \mathrm{nil}, \{\text{KNOWS}\}, \varnothing, (1, 1)) \ , \quad \rho_2 = (\rightarrow, \mathrm{nil}, \{\text{KNOWS}\}, \varnothing, (2, 2)) \ .$$

Consider again the property graph G in Figure 4. Let

$$p_1 = n_1 r_1 n_2 r_2 n_3 \qquad u_1 = \{\, x \mapsto n_1, y \mapsto n_3, z \mapsto n_2 \,\}$$
$$p_2 = n_1 r_1 n_2 r_2 n_3 r_3 n_4 \qquad u_2 = \{\, x \mapsto n_1, y \mapsto n_4, z \mapsto n_2 \,\}$$

Then, $(p_1, G, u_1) \models \pi_1$ and $(p_2, G, u_2) \models \pi_2$; therefore, π is satisfied in G by p_1 under u_1 and by p_2 under u_2. This shows the ability of a variable length pattern to match paths of varying length.

In addition, variable length patterns may admit several assignments even for a single given path. To see this, note that p_2 satisfies π in G *also* under the assignment u'_2 that agrees with u_2 on x and y but maps z to n_3, because $(p_2, G, u'_2) \models \pi_3$. □

In Cypher, we want to return the "matches" for a pattern in a graph, not simply check whether the pattern is satisfied (i.e., there exists a match). This is captured formally next.

Pattern matching. The set of *free variables* of a node pattern $\chi = (a, L, P)$, denoted by $\mathsf{free}(\chi)$, is $\{a\}$ whenever a is not nil, and empty otherwise. For a relationship pattern ρ, the set $\mathsf{free}(\rho)$ is defined analogously. Then, for a path pattern π we define $\mathsf{free}(\pi)$ to be union of all free variables of each node and relationship pattern occurring in it. For example, for the pattern π of Example 4.4 we have $\mathsf{free}(\pi) = \{x, y, z\}$. For named patterns, $\mathsf{free}(\pi/a) = \mathsf{free}(\pi) \cup \{a\}$. Then, for a path pattern π (optionally named), a graph G and an assignment u, we define

$$\mathsf{match}(\pi, G, u) = \biguplus_{\substack{p \text{ in } G \\ \pi' \in \mathsf{rigid}(\pi)}} \left\{ u' \;\middle|\; \begin{array}{l} \mathsf{dom}(u') = \mathsf{free}(\pi) - \mathsf{dom}(u) \\ \text{and } (p, G, u \cdot u') \models \pi' \end{array} \right\} \quad (1)$$

Note that, even though both u' and π' range over infinite sets, only a finite number of values contribute to a non-empty set in the final union. Thus $\mathsf{match}(\pi, G, u)$ is finite.

In (1), $\uplus$ stands for bag union: whenever a new combination of π' and p is found such that $(p, G, u \cdot u') \models \pi'$, a new occurrence of u' is added to $\mathsf{match}(\pi, G, u)$. This is in line with the way Cypher combines the **MATCH** clause and bag semantics, which is not captured by the satisfaction relation alone.

Example 4.5. Consider once again the graph G in Figure 4, and let π be the following variable length pattern:

```
(x:Teacher) -[:KNOWS*1..2]-> ()
            -[:KNOWS*1..2]-> (y:Teacher)
```

This is similar to the pattern in Example 4.4, but the middle node pattern is not given any name here: $\mathsf{free}(\pi) = \{x, y\}$. Indeed, $\mathsf{rigid}(\pi)$ is the same as in the previous example, with $\chi_2 = (\mathsf{nil}, \varnothing, \varnothing)$.

Let $p = n_1 r_1 n_2 r_2 n_3 r_3 n_4$ and $u = \{x \mapsto n_1, y \mapsto n_4\}$; it is easy to see that $(p, G, u) \models \pi_3 \in \mathsf{rigid}(\pi)$. However, observe that $(p, G, u) \models \pi_2$ as well (whereas π_1 and π_4 are not satisfied by any path of G). This shows that there may be multiple ways for a single path to satisfy a variable length pattern even under the same assignment. In our example, two copies of u will be added to $\mathsf{match}(\pi, G, \varnothing)$. □

Matching tuples of path patterns. Cypher allows one to match a tuple $\bar{\pi} = (\pi_1, \ldots, \pi_n)$ of path patterns, each optionally named. We say that $\bar{\pi}$ is rigid if all its components are rigid, and $\mathsf{rigid}(\bar{\pi})$ is defined as $\mathsf{rigid}(\pi_1) \times \cdots \times \mathsf{rigid}(\pi_n)$. The set of free variables of $\bar{\pi}$ is defined as $\mathsf{free}(\bar{\pi}) = \bigcup_{\pi_i} \mathsf{free}(\pi_i)$. Let $\bar{p} = (p_1, \ldots, p_n)$ be a tuple of paths; we write $(\bar{p}, G, u) \models \bar{\pi}$ if *no relationship id occurs in more than one path in* $\bar{p}$ and $(p_i, G, u) \models \pi_i$ for each $i \in \{1, \ldots, n\}$. Then, for a tuple of patterns $\bar{\pi}$, a graph G and an assignment u, $\mathsf{match}(\bar{\pi}, G, u)$ is defined as in (1), with the difference that the bag union is now over tuples $\bar{\pi}' \in \mathsf{rigid}(\pi)$ and $\bar{p}$ of paths.

Complexity. *Graph homomorphism* is the notion commonly used in graph querying [6, 9, 16]. At first, the pattern matching mechanism of Cypher might seem an extension of it that accounts for additional features such as values associated with nodes and relationships. However, on a more careful examination one may notice that Cypher actually departs from graph homomorphism. Indeed, in defining how rigid patterns are satisfied, we only looked at paths in which relationship ids occur *at most once*. In other words, a path cannot traverse the same relationship (i.e., edge) more than once.

Why do we have such a restriction, and how does it affect the complexity of pattern matching? Graph pattern matching is a canonical NP-complete problem: for a graph G and a pattern π, checking whether there is a match for π in G is NP-complete. Note that this is consistent with finding patterns in relational querying: for instance, given a join query Q and a relational database D, checking if $Q(D)$ returns a tuple is NP-complete as well [2].

However, Cypher goes beyond simple matching, by returning paths. This is where the problem occurs. Suppose we have a graph with a single node n a single relationship r, whose source and target are n, and let π be the pattern `(x)-[*0..]->(x)`. What would it return under homomorphism semantics? Since there are infinitely many paths from n to n, it appears that such a pattern will match *infinitely many times*, i.e., for each $m \geq 0$, it will produce a match that traverses the unique edge in the graph m times.

To avoid this situation, Cypher chose to disallow repeating relationship edges while matching patterns. In the above example, two matches will be returned: one for traversing the unique edge zero times, one for traversing it a single time. The question now is: how different is the complexity of this pattern matching?

By using the problem of the existence of two disjoint paths in a graph, one can prove that checking whether there is a match for π in G according to the Cypher semantics remains NP-complete. In fact, differently from the case of homomorphism-based matching, there is even a fixed path for which this is true. Nonetheless, such fixed patterns are of rather peculiar shape and they are not, based on many years of experience, a common occurrence in practice.

4.3 Formal Syntax and Semantics

We now present the key components of Cypher, namely *expressions*, *clauses*, and *queries*, and define their formal semantics. Together with pattern matching defined in the previous section, they will constitute the formalization of the core of Cypher.

The syntax of Cypher patterns was given in Figure 3. The syntax of Cypher expressions, clauses, and queries is provided in Figure 5.

Semantics of expressions. The semantics of an expression expr is a value $[\![\text{expr}]\!]_{G,u}$ in $\mathcal{V}$ determined by a property graph G and an assignment u that provides bindings for the names used in expr.

The rules here are fairly straightforward and given in full detail in [18]. Below we explain them briefly.

Values and variables. The semantics of $v \in \mathcal{V}$ is v itself, and the semantics of $a \in A$ is given by $u(a)$.

Maps. Here expr.k is the value associated with key k; {} is the empty map, and every prop_list, which is of the form $\{k_1 : e_1, \ldots, k_m : e_m\}$ (see Section 4.2) can be seen as a map.

Lists. These expressions either test if a value is in the list, or form a list from a sequence of values, or select elements and sublists of a list, given by their positions.

Strings. We have some basic operations on strings such as looking for prefix, suffix, and subword (starts-with, ends-with, contains).

Logic. Just like SQL, Cypher uses 3-value logic for dealing with nulls. The values are **true**, **false**, and **null** (unknown), and the rules for connectives and, or, not, and xor, are exactly the same as in SQL. Also one can test if a value is **null**.

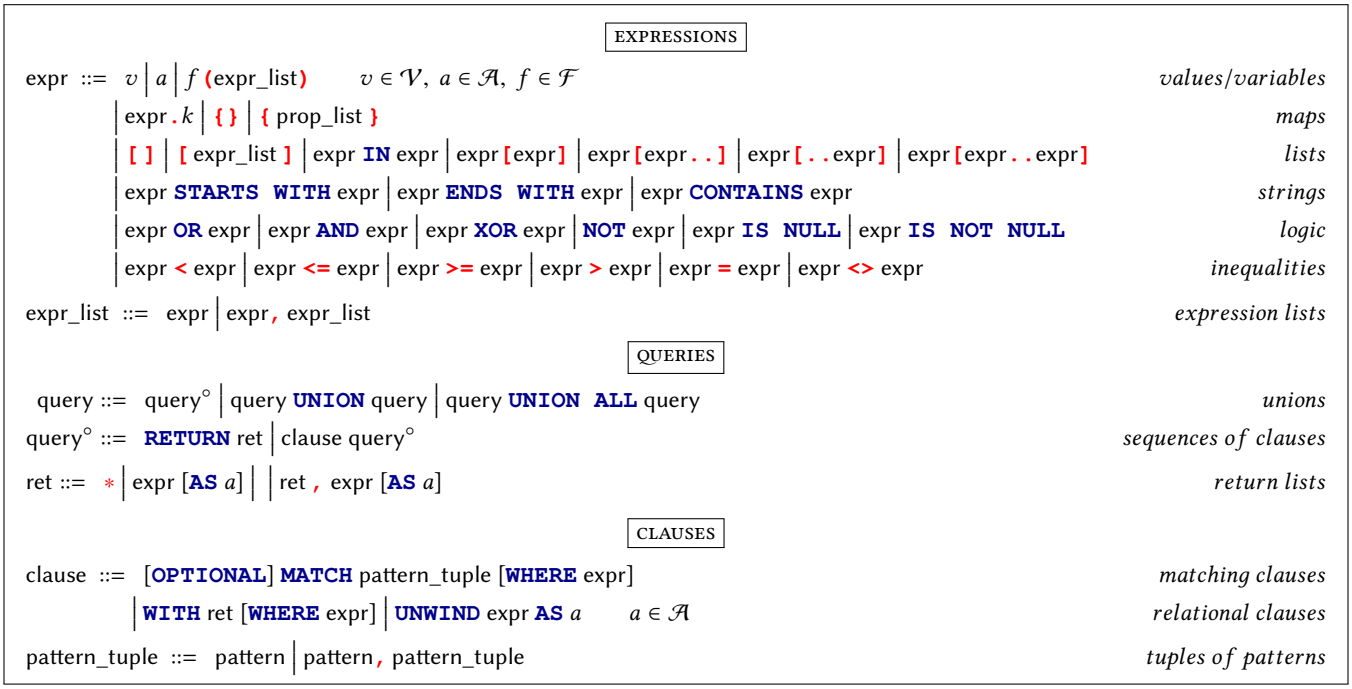

Figure 5: Syntax of expressions, queries, and clauses of core Cypher

Inequalities. Of course there are standard comparisons between numerical values.

Semantics of queries. A query is either a sequence of clauses ending with the **RETURN** statement, or a union (set of bag) of two queries. The **RETURN** statement contains the return list, which is either $*$, or a sequence of expressions, optionally followed by **AS** a, to provide their names.

To provide the semantics of queries, we assume that there exists an (implementation-dependent) injective function α that maps expressions to names. Recall that the semantics of both queries and clauses, relative to a property graph G, is a function from tables to tables, so we shall describe its value on a table T, i.e., $[\![\text{query}]\!]_G(T)$. The rules are provided in Figure 6.

In the figure we make the following assumptions. First, the fields of T are $b_1, \ldots, b_q$. Second, if we have a return list e_1 [**AS** a_1], $\ldots, e_m$ [**AS** a_m] with optional AS for some of the expressions, then $a'_i = a_i$ if **AS** a_i is present in the list, and $a'_i = \alpha(a_i)$ otherwise, with the added requirement that all the a'_is are distinct. In some rules for the semantics, some AS could be optional. It is assumed that when such optional AS is present on the left side, then it is also present on the right hand side. Finally, in Figure 6, Q, Q_1, Q_2 refer to queries and C to clauses.

Semantics of clauses. The meaning of Cypher clauses is again functions that take tables to tables. These are split into two classes. Matching clauses are essentially pattern matching statements: they are of the form **OPTIONAL MATCH** pattern_tuple **WHERE** expr. Both **OPTIONAL** and **WHERE** could be omitted. The key to their semantics is pattern matching, in particular $\text{match}(\bar{\pi}, G, u)$ described in Section 4.2 (see Equation 1).

Similarly to the description of the semantics of queries in Figure 6, we make the assumption that the fields of T are $b_1, \ldots, b_q$. Our convention about the names a'_i are exactly the same as for queries (see above), except that $a'_i = \alpha(e_i)$ only if e_i is a name.

The **MATCH** clause extends the set of field names of T by adding to it field names that correspond to names occurring in the pattern but not in u. It also adds tuples to T, based on matches of the pattern that are found in graphs. **UNWIND** is another clause that expands the set fields, and **WITH** clauses can change the set of fields to any desired one.

Example 4.6. Let G be the property graph defined in Figure 4. consider the clause **MATCH** π, where π is the pattern

```
(x) -[:KNOWS*]-> (y)
```

Let T be the table $\{(x : n_1); (x : n_3)\}$ with a single field x. We show how to compute $[\![\textbf{MATCH}\ \pi]\!]_G(T)$.

Note that $\text{rigid}(\pi)$ is the (infinite) set of all rigid paths $\pi_m = (\rightarrow, \text{nil}, \{\text{KNOWS}\}, m, m)$, for $m > 0$. These can only be satisfied by paths with exactly m distinct relationships. Since G only contains 3 relationships, only π_1, π_2 and π_3 can contribute to the result.

Let $u = (x : n_1)$, $\pi' = \pi_1$ and $p = n_1 r_1 n_2$. Then $\text{free}(\pi_1) - \text{dom}(u) = \{y\}$, and thus u' must be a record over the field y. One can easily check that $(n_1 r_1 n_2, G, (x : n_1, y : n_2)) \models \pi_1$. In fact n_2 is the only suitable value for y, and thus the contribution of this specific triple u, π', p to the final result is precisely $\{(x : n_1, y : n_2)\}$.

No path p other than $n_1 r_1 n_2$ can contribute a record in the case where $u = (x : n_1)$ and $\pi' = \pi_1$. Indeed, π_1 requires p to be of length 1, and start at x, which u evaluates to be n_1. By a similar reasoning, we can compute the contribution of the following triples:

- $(x : n_1, y : n_3)$, π_2, $n_1 r_1 n_2 r_2 n_3$ yields $(x : n_1, y : n_3)$;

$$
\begin{aligned}
[\![\texttt{RETURN}\ *]\!]_G(T) &= T \text{ if } T \text{ has a least one field}\\
[\![\texttt{RETURN}\ e_1\ [\texttt{AS}\ a_1], \ldots, e_m\ [\texttt{AS}\ a_m]]\!]_G(T) &= \biguplus_{u \in T} \{(a'_1 : [\![e_1]\!]_{G,u}, \ldots, a'_m : [\![e_m]\!]_{G,u})\}\\
[\![\texttt{RETURN}\ *, e_1\ [\texttt{AS}\ a_1], \ldots, e_m\ [\texttt{AS}\ a_m]]\!]_G(T) &= [\![\texttt{RETURN}\ b_1\ \texttt{AS}\ b_1, \ldots, b_q\ \texttt{AS}\ b_q, e_1\ [\texttt{AS}\ a_1], \ldots, e_m\ [\texttt{AS}\ a_m]]\!]_G(T)\\
[\![Q_1\ \texttt{UNION ALL}\ Q_2]\!]_G(T) &= [\![Q_1]\!]_G(T) \cup [\![Q_2]\!]_G(T)\\
[\![Q_1\ \texttt{UNION}\ Q_2]\!]_G(T) &= \varepsilon\Big([\![Q_1]\!]_G(T) \cup [\![Q_2]\!]_G(T)\Big)\\
[\![C\ Q]\!]_G(T) &= [\![Q]\!]_G\Big([\![C]\!]_G(T)\Big)
\end{aligned}
$$

Figure 6: Semantics of Cypher queries

$$
\begin{aligned}
[\![\texttt{MATCH}\ \bar{\pi}]\!]_G(T) &= \biguplus_{u \in T} \{u \cdot u' \mid u' \in \mathsf{match}(\bar{\pi}, G, u)\}\\
[\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ \mathrm{expr}]\!]_G(T) &= [\![\texttt{WHERE}\ \mathrm{expr}]\!]\Big([\![\texttt{MATCH}\ \bar{\pi}]\!]_G(T)\Big)\\
[\![\texttt{OPTIONAL MATCH}\ \bar{\pi}\ \texttt{WHERE}\ \mathrm{expr}]\!]_G(T) &= \biguplus_{u \in T} \begin{cases} [\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ \mathrm{expr}]\!]_G(\{u\}) & \text{if } [\![\texttt{MATCH}\ \bar{\pi}\ \texttt{WHERE}\ \mathrm{expr}]\!]_G(\{u\}) \neq \varnothing \\ (u, (\mathrm{free}(u, \bar{\pi}) : \texttt{null})) & \text{otherwise} \end{cases}\\
[\![\texttt{OPTIONAL MATCH}\ \bar{\pi}]\!]_G(T) &= [\![\texttt{OPTIONAL MATCH}\ \bar{\pi}\ \texttt{WHERE true}]\!]_G(T)\\
[\![\texttt{WITH}\ *]\!]_G(T) &= T \text{ if } T \text{ has a least one field}\\
[\![\texttt{WHERE}\ \mathrm{expr}]\!]_G(T) &= \{u \in T \mid [\![\mathrm{expr}]\!]_{G,u} = \texttt{true}\}\\
[\![\texttt{WITH}\ e_1\ [\texttt{AS}\ a_1], \ldots, e_m\ [\texttt{AS}\ a_m]]\!]_G(T) &= \biguplus_{u \in T} \{(a'_1 : [\![e_1]\!]_{G,u}, \ldots, a'_m : [\![e_m]\!]_{G,u})\}\\
[\![\texttt{WITH}\ *, e_1\ [\texttt{AS}\ a_1], \ldots, e_m\ [\texttt{AS}\ a_m]]\!]_G(T) &= [\![\texttt{WITH}\ b_1\ \texttt{AS}\ b_1, \ldots, b_q\ \texttt{AS}\ b_q, e_1\ [\texttt{AS}\ a_1], \ldots, e_m\ [\texttt{AS}\ a_m]]\!]_G(T)\\
[\![\texttt{WITH}\ \mathrm{ret}\ \texttt{WHERE}\ \mathrm{expr}]\!]_G(T) &= [\![\texttt{WHERE}\ \mathrm{expr}]\!]_G\Big([\![\texttt{WITH}\ \mathrm{ret}]\!]_G(T)\Big)\\
[\![\texttt{UNWIND}\ \mathrm{expr}\ \texttt{AS}\ a]\!]_G(T) &= \biguplus_{u \in T} \biguplus_{v \in E_u} \{(u, a : v)\}, \quad \text{where } E_u = \begin{cases} \biguplus_{0 \le i < m} \{v_i\} & \text{if } [\![\mathrm{expr}]\!]_{G,u} = \mathsf{list}(v_0, \ldots, v_{m-1}) \\ \{\} & \text{if } [\![\mathrm{expr}]\!]_{G,u} = \mathsf{list}() \\ \{[\![\mathrm{expr}]\!]_{G,u}\} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 7: Semantics of Cypher clauses

- $(x : n_1, y : n_4)$, π_3, $n_1 r_1 n_2 r_2 n_3 r_3 n_4$ yields $(x : n_1, y : n_4)$;
- $(x : n_3, y : n_4)$, π_1, $n_3 r_3 n_4$ yields $(x : n_3, y : n_4)$;

and show that the contributions of all other possible combinations of records, paths and patterns are empty. This tells us that $[\![\texttt{MATCH}\ \pi]\!]_G(T)$ is the table with the rows $(x : n_1, y : n_2)$, $(x : n_1, y : n_3)$, $(x : n_1, y : n_4)$, and $(x : n_3, y : n_4)$.

5 HISTORICAL REMARKS

The Cypher query language emerged from the evolution of the Neo4j graph database, which in turn originated from a data model that was first conceived of in 2000 by the founders of Neo4j in the course of building a media asset management system. The system's data model changed frequently, and had complex data structures and access control views which inspired the idea of tagging network elements with 'captions' or property sets. The high emphasis on relationship information (modeled as graph edges) coupled with the need to support variable-length path traversals led away from the schema-rigid relational database initially used, to the creation of a *native* property graph database system.

From 2007 onwards this technology began to be provided for general use as an open-source database management system. Initially Neo4j was embedded as a Java library: the Tinkerpop Blueprints and Gremlin APIs, now part of the Apache project, originated in the early incarnations of Neo4j. Increased usage led to the inception of a property graph query language which would occupy the same ecological niche as SQL. The development of this language went hand in hand with changes in the database implementation that increasingly automated search optimizations through indexing of node data, which in turn drove the addition of node labels to the original scheme of relationship types.

From these influences emerged the current Cypher data model and the predominance of the Cypher language, which became the primary way of interacting with the graph. Cypher was largely an invention by Andrés Taylor, engineer at Neo4j and co-author of this paper, in early 2011, and is at a syntactic and feature level inspired by SQL, and also incorporates concepts from functional programming,

Python and SPARQL. There is a strong family relationship to the feature set of Gremlin, including the concept of a linear flow of successive data operations, which gives Cypher a form of functional composition which is quite different in feel to the nesting structure of subqueries in SQL.

In late 2015 Neo4j announced the openCypher project, providing an open platform to drive the standardization of Cypher as *the* property graph query language. The openCypher project has published a number of artefacts under the Apache 2.0 license, including EBNF and ANTLR4 grammars and a Technology Compatibility Kit (TCK), designed using a language neutral framework (Cucumber).

Proposals for language change are open, allowing anyone to participate in the design of Cypher, and during 2017 a series of public meetings between Cypher implementers has provided a forum for discussing language changes and agreeing on language extensions. It was recently agreed that the current state of the Cypher language should be referred to as Cypher 9, which we have presented thus far in this paper.

6 CURRENT DEVELOPMENTS

A consensus-driven openCypher Implementers Group (oCIG) now governs the evolution of the Cypher language. For most of 2017 it has been working on the definition of Cypher 10, which will be described in a complete natural language specification, combined with an extension of the formal semantics presented here. Three new features are being added to this next version of the language: multiple graphs, query composition, and temporal data types (e.g. date-time). These reflect industrial usage experience of Cypher, particularly in Neo4j and SAP HANA Graph [50].

Multiple graphs. Cypher 9 assumes an implicit single global graph that is used both for querying and for updating operations. Applications sometimes work with multiple disconnected subcomponents of this global graph. Such components may be understood as distinct property graphs themselves. However, Cypher does not yet include a mechanism for referencing such graphs explicitly and therefore the language does not easily allow operations to be applied to specific graphs.

The Cypher 10 proposal for multiple graphs introduces named graph references, which represent externally located graphs, graphs created by the query, or graphs created by a previous query in a composition of queries. Graph references may be passed as arguments to, and returned as results from, Cypher 10 queries, and can be used in set operations. These capabilities broadly match comparable functionality already present in SPARQL 1.1 [58]. Named graphs contribute to natural partitioning and graph transformations (which in turn enable graph-compositional queries).

Query composition. Passing multiple graph references requires extending the existing tabular composition mechanism in Cypher 9, where the `WITH` projection clause turns the output of a query into an input driving table for a following query. The Cypher 10 proposal for multiple graphs adds the ability to instead pass a "table-graphs" construct, consisting of a single table and multiple named graphs as query arguments. A table-graphs may additionally pass information relating to which of these graphs is being used for reading (source graph) and updating operations (target graph). Similarly a query result is a table-graphs. This enables Cypher queries to be composed as a chain of elementary queries. With the addition of subqueries such query chains can also be formed into a tree.

This parallels the advocacy of a similar feature by the LDBC Query Language Task Force in their description of the G-CORE research language [5]. The two proposals differ in that G-CORE describes queries that output only a single graph. In Cypher the projection of tabular results is recognized as necessary for applications to access property values, and consequently Cypher 10 is closed under table-graphs.

As part of these new query composition features, Cypher 10 also introduces *named queries*. Named queries simplify the creation of libraries of re-usable queries which can be composed in different query trees. They also form the basis for offering graph views.

Example 6.1. The following graph transformation query first finds all pairs of persons that have at least a single friend in common and then returns a new graph where they are connected directly:

```
FROM GRAPH soc_net AT "hdfs://.../soc_network"
MATCH (a)-[r1:FRIEND]-()-[r2:FRIEND]-(b)
WHERE abs(r2.since-r1.since) < $duration
WITH DISTINCT a, b
RETURN GRAPH friends OF (a)-[:SHARE_FRIEND]->(b)
```

The result of this query may then be composed with a follow-up query, e.g. for further filtering for friend-sharing-friends that live in the same city:

```
QUERY GRAPH friends
MATCH (a)-[:SHARE_FRIEND]-(b)
FROM GRAPH register AT "bolt://.../citizens"
MATCH (a)-[:IN]->(c:City)<-[:IN]-(b)
RETURN *
```

Drafts of the Cypher 10 proposal for compositional graph queries have already been implemented in Cypher for Apache Spark.

Temporal types. A detailed proposal[3] specifies support for temporal instant types (DateTime, LocalDateTime, Date, Time, and LocalTime) and a duration type.

7 RELATED WORK

Graph data models. These in general have been a topic of research since the 1990s. We provide here a summary and refer the reader to the surveys [6, 7] for further reading.

Variations of simple directed labeled graphs are presented in [4, 23, 24, 26], and work undertaken on the hypergraph data model is detailed in [28, 35–37, 51]. Semi-structured data models closely related to graph data models include XML [10] and OEM [48].

RDF [59], a W3C recommendation, models resources using a graph model, and forms the foundation of the Semantic Web. Its basic building block is a triple, consisting of a *subject*, which describes the resource and is modeled as a node; a *predicate* which is a property of the resource and is modeled as an edge; and an *object*, which is the value of the property. A triple is therefore a statement of the relationship between the subject and the object, and a set of triples forms a graph. In contrast to the property graph model, RDF only supports a single value on a node or edge.

[3] https://github.com/thobe/openCypher/blob/date-time/cip/CIP2015-08-06-date-time.adoc

Graph query languages. Some graph navigation can be expressed in SQL using recursive queries. Basic properties such as reachability in a graph can be expressed fairly easily, but complex graph patterns that involve both data and navigation become very cumbersome using SQL's recursion. In addition, some features of Cypher, such as non-repeatability of edges in pattern matching, cannot in general be expressed in SQL. Deficiencies of SQL as a graph query language led to a large body of research on proper graph query languages. We briefly summarize the main contributions here and refer the reader to comprehensive surveys [6–8, 60].

Regular path queries (RPQs) – the ability to express a path between any two nodes as a regular expression over the edge labels – were first proposed by in 1987 in [16], and extended to conjunctive RPQs by means of taking joins in [14]; these classes of queries form the basis of languages studied in the theoretical literature [8, 60]. RPQs and related languages were extended to a theoretical model of data graphs that resemble property graphs in [38], which also described `GXPath`, a graph extension of XPath with node tests.

SPARQL [58] is the standard language used for querying RDF data [59], and support for RPQs was added in SPARQL 1.1 through the introduction of *property paths* [33].

There are other industrial property graph query languages. Oracle's PGQL [56] uses the **`SELECT–FROM–WHERE`** form from SQL and the graph pattern syntax from Cypher. It allows for RPQs through the use of the **`PATH`** clause defining a regular path pattern. It does not support data insertion or updating. Gremlin [53] is a property graph dataflow language designed by Apache Tinkerpop, which supports procedural pattern matching and traversals embedded in general purpose languages, but does not allow independent statement of a declarative query in the style of SQL and Cypher. The U.S. standards body INCITS recently created a working group to examine potential extensions to the SQL standard to allow relational data to be viewed and queried as property graph data.

8 FUTURE WORK

Major topics under discussion for language changes beyond the current developments around Cypher 10 include: richer path patterns, allowing selection of additional semantics of pattern matching (homomorphism, node isomorphism, edge isomorphism), enlarging the schema model, expanding the underlying data model, and clarifying theoretical vs real-life complexity of Cypher queries.

Path patterns. Cypher 9 supports a more limited form of regular path queries than for example SPARQL [58] and does not support full regular expression composition. A future version of Cypher will add an extended form of regular path patterns that allows data tests on nodes and relationships and path cost declarations. In addition, it will be possible to define named path patterns which can be referenced in other path patterns. This latter change was first proposed in PGQL [56].

Configurable morphisms. Cypher 9 matches patterns using relationship (edge) isomorphism, i.e. each matched instance of a given pattern never binds the same relationship from the underlying data graph to more than one relationship variable or path variable. This restriction reduces the number of pattern matches and ensures that variable length relationship patterns never produce infinite result sets. In some applications this rule is insufficient: it is envisioned that Cypher will also allow the writer of a query to specify the use of homomorphism or node isomorphism pattern matching semantics.

Schema model. Cypher was originally conceived in a dynamically typed, schema-less context. Neo4j nowadays is schema-optional, i.e. it supports an additional schema constraint language (e.g. for requiring nodes with a given label to have certain properties). Other implementations of Cypher assume a more strict schema (e.g., Cypher for Apache Spark, or SAP HANA Graph). Standardizing a schema model will improve data modeling capabilities and help portability of queries while also enabling implementations to type check queries more rigidly up-front as well as optimize them better.

Data model. Current industry trends indicate the need for supporting both spatial and temporal (time-evolving) graph data as well as stream processing applications. These trends point towards future evolutions of the core data model and underlying type system together with other related language changes.

Theoretical vs real-life complexity. We have seen that pattern matching under the Cypher semantics can be NP-complete, even for a fixed pattern, and thus unlikely to be tractable in general. Theoretically, this could be viewed as a reason to look for an alternative semantics. However, in practice this works well: people do not write queries that look like reductions from NP-complete problems. While in databases, high theoretical complexity has usually been seen as a show stopper, in other fields this is not so: very fast industrial strength SAT solvers routinely solve NP-complete problems of large size [40], typechecking in some popular languages could provably take exponential time in the worst case [31], and problems of astronomical complexity or even undecidable ones have been successfully tackled [15, 32]. The reason is the same as in the case of Cypher queries: examples from the worst case analysis very rarely arise in practice. This observation leads to a new research program on analyzing real-life practical complexity of queries and its differences with the theoretical complexity of languages.

9 CONCLUSION

The property graph data model is increasingly prevalent across a wide variety of application domains, where data is represented naturally as a graph structure and there is a requirement for the query language to allow for graph-oriented operations (e.g., transitive closure) to be expressed *directly*. This has also fueled the use of native graph data structures to optimize storage and query processing for such operations.

Cypher is a solidly-established declarative query language for the property graph model, with increasing adoption in multiple products and projects. The language is evolving with new features – saliently support for multiple graphs and query composition. Under the aegis of the openCypher Implementers Group it is now being documented as a fully-specified standard that can be independently implemented using different architectures, and varying storage and query optimization strategies.

All this work – including the formal semantics described in this paper – will contribute towards the goal of establishing Cypher alongside SQL as complementary languages, enabling integrated use of the graph and relational data models.

REFERENCES

[1] H. Abelson et al. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] D. Alocci, J. Mariethoz, O. Horlacher, J. T. Bolleman, M. P. Campbell, and F. Lisacek. Property graph vs RDF triple store: A comparison on glycan substructure search. *PLOS ONE*, 10(12):1–17, 12 2015.

[4] B. Amann and M. Scholl. Gram: a graph data model and query languages. In *Proceedings of the ACM conference on Hypertext*, pages 201–211. ACM, 1992.

[5] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutiérrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-CORE A Core for Future Graph Query Languages. In *ACM SIGMOD*, 2018.

[6] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, Sept. 2017.

[7] R. Angles and C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.

[8] P. Barceló. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 175–188, 2013.

[9] P. Barceló, L. Libkin, and J. L. Reutter. Querying regular graph patterns. *Journal of the ACM*, 61(1):8:1–8:54, 2014.

[10] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[11] M. Cabra. How the ICIJ used Neo4j to unravel the Panama Papers. Neo4j Blog, May 2016. https://neo4j.com/blog/icij-neo4j-unravel-panama-papers/.

[12] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[13] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 510–524. ACM, 2017.

[14] M. Consens and A. Mendelzon. Graphlog: A visual formalism for real life recursion. In *9th ACM Symposium on Principles of Database Systems (PODS)*, pages 404–416, 1990.

[15] B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011.

[16] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *SIGMOD Conference*, pages 323–330. ACM Press, 1987.

[17] G. Drakopoulos, A. Kanavos, and A. K. Tsakalidis. Evaluating twitter influence ranking with system theory. In *Proceedings of the 12th International Conference on Web Information Systems and Technologies, WEBIST 2016, Volume 1, Rome, Italy, April 23-25, 2016*, pages 113–120, 2016.

[18] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, M. Schuster, P. Selmer, and A. Taylor. Formal semantics of the language cypher. *CoRR*, abs/1802.09984, 2018. https://arxiv.org/abs/1802.09984.

[19] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 209–218. IEEE, 1993.

[20] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1):27–39, 2017.

[21] A. Gubichev. *Query Processing and Optimization in Graph Databases*. PhD thesis, Technical University Munich, 2015.

[22] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In *Computer Science Logic*, pages 274–308, 1992.

[23] R. H. Güting. Graphdb: Modeling and querying graphs in databases. In *VLDB*, pages 297–308. Morgan Kaufmann, 1994.

[24] M. Gyssens, J. Paredaens, J. V. den Bussche, and D. V. Gucht. A graph-oriented object database model. *IEEE Trans. Knowl. Data Eng.*, 6(4):572–586, 1994.

[25] N. Hawes, B. Barham, and C. Cifuentes. FrappÉ: Querying the linux kernel dependency graph. In *Proceedings of the GRADES'15*, GRADES'15, pages 4:1–4:6. ACM, 2015.

[26] J. Hidders. Typing graph-manipulation operations. In *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 391–406. Springer, 2003.

[27] H. H. Huang and H. Liu. Big data machine learning and graph analytics: Current state and future challenges. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 16–17, 2014.

[28] B. Iordanov. Hypergraphdb: A generalized graph database. In *WAIM Workshops*, volume 6185 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2010.

[29] M. Junghanns, A. Petermann, K. Gómez, and E. Rahm. GRADOOP: scalable graph data management and analytics with hadoop. *CoRR*, abs/1506.00548, 2015.

[30] M. Kay. *XPath 2.0 programmer's reference*. John Wiley & Sons, 2004.

[31] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, 1994.

[32] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586, 2002.

[33] E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoč. SPARQL with property paths. In *The Semantic Web - ISWC 2015*, pages 3–18, 2015.

[34] J. Larriba-Pey, N. Martínez-Bazan, and D. Domínguez-Sal. Introduction to graph databases. In *Reasoning Web*, volume 8714 of *Lecture Notes in Computer Science*, pages 171–194. Springer, 2014.

[35] M. Levene and G. Loizou. A graph-based data model and its ramifications. *IEEE Trans. Knowl. Data Eng.*, 7(5):809–823, 1995.

[36] M. Levene and A. Poulovassilis. The hypernode model and its associated query language. In *Jerusalem Conference on Information Technology*, pages 520–530. IEEE, 1990.

[37] M. Levene and A. Poulovassilis. An object-oriented data model formalised through hypergraphs. *Data Knowl. Eng.*, 6:205–234, 1991.

[38] L. Libkin, W. Martens, and D. Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016.

[39] A. Lysenko, I. A. Roznovat, M. Saqi, A. Mazein, C. J. Rawlings, and C. Auffray. Representing and querying disease networks using graph databases. *BioData Mining*, 9(1):23, Jul 2016.

[40] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[41] J. Marton, G. Szárnyas, and M. Búr. Model-driven engineering of an opencypher engine: Using graph queries to compile graph queries. In *SDL Forum*, volume 10567 of *Lecture Notes in Computer Science*, pages 80–98. Springer, 2017.

[42] R. Milner, M. Tofte, and R. Harper. *Definition of Standard ML*. MIT Press, 1990.

[43] J. C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[44] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 539–552. ACM, 2008.

[45] J. Mullen, S. J. Cockell, P. Woollard, and A. Wipat. An integrated data driven approach to drug repositioning using gene-disease associations. *PLOS ONE*, 11(5):1–24, 05 2016.

[46] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[47] openCypher. Cypher Query Language Reference, Version 9, Nov. 2017. https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf.

[48] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260. IEEE Computer Society, 1995.

[49] N. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, NTUA, 253pp, 1998.

[50] M. Paradies. Graph pattern matching in SAP HANA. First openCypher Implementers Meeting, Feb. 2017. https://tinyurl.com/ycxu54pr.

[51] A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Trans. Inf. Syst.*, 12(1):35–68, 1994.

[52] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. O'Reilly Media, 2013.

[53] M. A. Rodriguez. The Gremlin graph traversal machine and language. In *DBPL*, pages 1–10. ACM, 2015.

[54] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[55] B. A. Steer, A. Alnaimi, M. A. B. F. G. Lotz, F. Cuadrado, L. M. Vaquero, and J. Varvenne. Cytosm: Declarative property graph queries without data migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, pages 4:1–4:6, 2017.

[56] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *GRADES*, page 7. ACM, 2016.

[57] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 425–446, 2010.

[58] W3C. *SPARQL 1.1 Query Language*, 2013. http://www.w3.org/TR/sparql11-query/.

[59] W3C. *RDF 1.1 Concepts and Abstract Syntax*, Sept. 2014. http://www.w3.org/TR/rdf11-concepts/.

[60] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[61] B.-H. Yoon, S.-K. Kim, and S.-Y. Kim. Use of graph database for the integration of heterogeneous biological data. *Genomics & informatics*, pages 19–27, 03 2017.

A APPENDIX: FULL SEMANTICS OF CYPHER EXPRESSIONS

We now provide details of the semantics of Cypher expressions. Assume that we are given a fixed property graph $G = (N, R, s, t, \iota, \lambda, \tau)$ and a fixed record $u = (a_1 : v_1, \ldots, a_n : v_n)$ that associates values $v_1, \ldots, v_n$ with names $a_1, \ldots, a_n$.

Values and variables.

- $[\![v]\!]_{G,u} = v$, where v is a value.
- $[\![a]\!]_{G,u} = u(a)$, where a is a name that belongs to the domain of u.
- $[\![f(e_1, \ldots, e_m)]\!]_{G,u} = f([\![e_1]\!]_{G,u}, \ldots, [\![e_m]\!]_{G,u})$, where $e_1, \ldots, e_m$ are expressions, and f is any m-ary function in $\mathcal{F}$ from values to values.

Maps.

- $[\![\text{expr}.k]\!]_{G,u} = \begin{cases} \iota([\![\text{expr}]\!]_{G,u}, k) & \text{if } [\![\text{expr}]\!]_{G,u} \in \mathcal{N} \cup \mathcal{R} \\ w_i & \text{if } [\![\text{expr}]\!]_{G,u} = \{k_1 : w_1, \ldots, k_m : w_m\} \text{ and } k = k_i \\ \texttt{null} & \text{if } [\![\text{expr}]\!]_{G,u} = \{k_1 : w_1, \ldots, k_m : w_m\} \text{ and } k \notin \{k_1, \ldots, k_m\} \\ & \text{or } [\![\text{expr}]\!]_{G,u} = \{\} \\ & \text{or } [\![\text{expr}]\!]_{G,u} = \texttt{null} \end{cases}$
  where k and the k_is are property keys, and w and the w_is are values.
- $[\![\{k_1 : e_1, \ldots, k_m : e_m\}]\!]_{G,u} = \text{map}((k_1, [\![e_1]\!]_{G,u}), \ldots, (k_m, [\![e_m]\!]_{G,u}))$
  where $k_1, \ldots, k_m$ are distinct property keys and $e_1, \ldots, e_m$ are expressions.
- $[\![\{k_1 : e_1, \ldots, k_m : e_m\}]\!]_{G,u} = [\![\{k_{i_1} : e_{i_1}, \ldots, k_{i_\ell} : e_{i_\ell}\}]\!]_{G,u}$
  where $k_1, \ldots, k_m$ are property keys, $e_1, \ldots, e_m$ are expressions, and $i_1, \ldots, i_\ell$ are distinct indices such that $\{k_{i_1}, \ldots, k_{i_\ell}\} = \{k_1, \ldots, k_m\}$ and for each p such that $i_p < m$, $k_{i_p} \notin \{k_{i_p+1}, \ldots, k_m\}$. In other words, if there are repeated keys among $k_1, \ldots, k_m$, only the last occurrence of each key is kept.
- $[\![\{\}]\!]_{G,u} = \text{map}()$

Lists.

- $[\![[e_1, \ldots, e_m]]\!]_{G,u} = \text{list}([\![e_1]\!]_{G,u}, \ldots, [\![e_m]\!]_{G,u})$
  where $e_1, \ldots, e_m$ are expressions.
- $[\![[\,]]\!]_{G,u} = \text{list}()$

Non-empty lists. Assume that expr is an expression such that $[\![\text{expr}]\!]_{G,u} = \text{list}(w_0, \ldots, w_{m-1})$ for some values $w_0, \ldots, w_{m-1}$. Then the semantics of list expressions is as follows.

- $[\![\text{expr}[\text{expr}']]\!]_{G,u} = \begin{cases} w_i & \text{if } 0 \le i < m \\ w_{m+i} & \text{if } -m \le i < 0 \\ \texttt{null} & \text{if } i < -m \text{ or } i \ge m \end{cases}$
  where $[\![\text{expr}']\!]_{G,u} = i$ for some integer i.
- $[\![\text{expr}[e_1..e_2]]\!]_{G,u} = \begin{cases} \text{list}(w_{\max(0,i')}, \ldots, w_{\min(m-1,j'-1)}) & \text{if } i' \le j', i' < m, j' > 0 \\ \text{list}() & \text{otherwise} \end{cases}$
  where $[\![e_1]\!]_{G,u} = i$ for some integer i, $[\![e_2]\!]_{G,u} = j$ for some integer j, $i' = i$ if $i \ge 0$ and $i' = m + i$ otherwise, $j' = j$ if $j \ge 0$ and $j' = m + j$ otherwise.
- $[\![\text{expr}[e_1..]]\!]_{G,u} = [\![\text{expr}[e_1..m]]\!]_{G,u}$
- $[\![\text{expr}[..e_2]]\!]_{G,u} = [\![\text{expr}[0..e_2]]\!]_{G,u}$
- $[\![\text{expr}' \texttt{ IN } \text{expr}]\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}' = w_i]\!]_{G,u} = \texttt{true}, \text{ for some integer } i,\ 0 \le i < n \\ \texttt{null} & \text{if the previous case does not hold} \\ & \text{and } [\![\text{expr}' = w_i]\!]_{G,u} = \texttt{null}, \text{ for some integer } i,\ 0 \le i < n \\ \texttt{false} & \text{otherwise} \end{cases}$

Empty lists. Assume that expr is an expression such that $[\![\text{expr}]\!]_{G,u} = \text{list}()$. Then the semantics of list expressions is as follows.

- $[\![\text{expr}[\text{expr}']]\!]_{G,u} = \texttt{null}$, where $[\![\text{expr}']\!]_{G,u} = i$ for some integer i.
- $[\![\text{expr}[e_1..e_2]]\!]_{G,u} = \text{list}()$
  where $[\![e_1]\!]_{G,u} = i$ for some integer i and $[\![e_2]\!]_{G,u} = j$ for some integer j.
- $[\![\text{expr}[e_1..]]\!]_{G,u} = \text{list}()$
  where $[\![e_1]\!]_{G,u} = i$ for some integer i.
- $[\![\text{expr}[..e_2]]\!]_{G,u} = \text{list}()$
  where $[\![e_2]\!]_{G,u} = j$ for some integer j.
- $[\![\text{expr}' \texttt{ IN } \text{expr}]\!]_{G,u} = \texttt{false}$ where expr′ is an expression.

Strings.

- $[\![\text{expr } \texttt{STARTS WITH } \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } \exists s, [\![\text{expr}]\!]_{G,u} = [\![\text{expr}']\!]_{G,u} \cdot s \\ \texttt{null} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{null} \text{ or } [\![\text{expr}']\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{otherwise} \end{cases}$

  where expr and expr′ are expressions such that $[\![\text{expr}]\!]_{G,u}$ and $[\![\text{expr}']\!]_{G,u}$ are either strings or **null**.
- $[\![\text{expr } \texttt{ENDS WITH } \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } \exists s, [\![\text{expr}]\!]_{G,u} = s \cdot [\![\text{expr}']\!]_{G,u} \\ \texttt{null} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{null} \text{ or } [\![\text{expr}']\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{otherwise} \end{cases}$

  where expr and expr′ are expressions such that $[\![\text{expr}]\!]_{G,u}$ and $[\![\text{expr}']\!]_{G,u}$ are either strings or **null**.
- $[\![\text{expr } \texttt{CONTAINS } \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } \exists s_1, s_2, [\![\text{expr}]\!]_{G,u} = s_1 \cdot [\![\text{expr}']\!]_{G,u} \cdot s_2 \\ \texttt{null} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{null} \text{ or } [\![\text{expr}']\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{otherwise} \end{cases}$

  where expr and expr′ are expressions such that $[\![\text{expr}]\!]_{G,u}$ and $[\![\text{expr}']\!]_{G,u}$ are either strings or **null**.

Logic.

Assume that expr and expr′ are expressions such that $[\![\text{expr}]\!]_{G,u} \in \{\texttt{true}, \texttt{false}, \texttt{null}\}$ and $[\![\text{expr}']\!]_{G,u} \in \{\texttt{true}, \texttt{false}, \texttt{null}\}$.

- $[\![\text{expr } \texttt{OR } \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{true} \text{ or } [\![\text{expr}']\!]_{G,u} = \texttt{true} \\ \texttt{false} & \text{if } [\![\text{expr}]\!]_{G,u} = [\![\text{expr}']\!]_{G,u} = \texttt{false} \\ \texttt{null} & \text{otherwise} \end{cases}$
- $[\![\text{expr } \texttt{AND } \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}]\!]_{G,u} = [\![\text{expr}']\!]_{G,u} = \texttt{true} \\ \texttt{false} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{false} \text{ or } [\![\text{expr}']\!]_{G,u} = \texttt{false} \\ \texttt{null} & \text{otherwise} \end{cases}$
- $[\![\text{expr } \texttt{XOR } \text{expr}']\!]_{G,u} = \begin{cases} \texttt{null} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{null} \text{ or } [\![\text{expr}']\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{if } [\![\text{expr}]\!]_{G,u} = [\![\text{expr}']\!]_{G,u} \text{ and } [\![\text{expr}]\!]_{G,u} \neq \texttt{null} \\ \texttt{true} & \text{otherwise} \end{cases}$
- $[\![\texttt{NOT } \text{expr}]\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{false} \\ \texttt{false} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{true} \\ \texttt{null} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{null} \end{cases}$

Value Comparisons.

Nulls. The rules follow SQL: in an expression, if an argument is null, then the value of the expression is null. The semantics of `IS NULL` is also the same as for SQL.

- $[\![\text{expr} \star \text{expr}']\!]_{G,u} = \texttt{null}$ if either $[\![\text{expr}]\!]_{G,u} = \texttt{null}$ or $[\![\text{expr}']\!]_{G,u} = \texttt{null}$, for $\star \in \{<, <=, >=, >, =, <>\}$.
- $[\![\text{expr } \texttt{IS NULL}]\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{null} \\ \texttt{false} & \text{if } [\![\text{expr}]\!]_{G,u} \neq \texttt{null} \end{cases}$
- $[\![\text{expr } \texttt{IS NOT NULL}]\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}]\!]_{G,u} \neq \texttt{null} \\ \texttt{false} & \text{if } [\![\text{expr}]\!]_{G,u} = \texttt{null} \end{cases}$

Empty maps. Assume that both expr and expr′ are expressions such that both $[\![\text{expr}]\!]_{G,u}$ and $[\![\text{expr}']\!]_{G,u}$ are maps, and one of them is map().

- $[\![\text{expr} = \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}]\!]_{G,u} = [\![\text{expr}']\!]_{G,u} = \text{map}() \\ \texttt{false} & \text{otherwise} \end{cases}$

Non-empty maps, same number of keys. Assume that $[\![\text{expr}]\!]_{G,u} = \{k_1 : w_1, \ldots, k_m : w_m\}$ and $[\![\text{expr}']\!]_{G,u} = \{k'_1 : w'_1, \ldots, k'_m : w'_m\}$, where $k_1, \ldots, k_m, k'_1, \ldots, k'_m$ are keys, and $w_1, \ldots, w_m, w'_1, \ldots, w'_m$ are values, and $m \geq 1$.

- $[\![\text{expr} = \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } \{k_1, \ldots, k_m\} = \{k'_1, \ldots, k'_m\} \\ & \quad \text{and } [\![\text{expr}.k_i = \text{expr}'.k_i]\!]_{G,u} = \texttt{true} \text{ for all } i \leq m \\ \texttt{null} & \text{if } \{k_1, \ldots, k_m\} = \{k'_1, \ldots, k'_m\} \\ & \quad \text{and } [\![\text{expr}.k_i = \text{expr}'.k_i]\!]_{G,u} = \texttt{null} \text{ for some } i \leq m \\ & \quad \text{and } [\![\text{expr}.k_i = \text{expr}'.k_i]\!]_{G,u} \neq \texttt{false} \text{ for all } i \leq m \\ \texttt{false} & \text{otherwise} \end{cases}$

Non-empty maps, different number of keys. Assume that $[\![\text{expr}]\!]_{G,u} = \{k_1 : w_1, \ldots, k_m : w_m\}$ and $[\![\text{expr}']\!]_{G,u} = \{k'_1 : w'_1, \ldots, k'_l : w'_l\}$, where $k_1, \ldots, k_m, k'_1, \ldots, k'_l$ are keys, and $w_1, \ldots, w_m, w'_1, \ldots, w'_l$ are values, $m, l \geq 1$, and $m \neq l$. In this case, $[\![\text{expr} = \text{expr}']\!]_{G,u} = \texttt{false}$.

Lists. Assume that both expr and expr′ are expressions such that both $[\![\text{expr}]\!]_{G,u}$ and $[\![\text{expr}']\!]_{G,u}$ are list values.

- $[\![\text{expr} = \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}]\!]_{G,u} = [\![\text{expr}']\!]_{G,u} = \text{list}() \\ & \text{or } [\![\text{expr}]\!]_{G,u} = [w_1, \ldots, w_m], [\![\text{expr}']\!]_{G,u} = [w'_1, \ldots, w'_m] \\ & \text{with } \forall i, [\![w_i = w'_i]\!]_{G,u} = \texttt{true} \\ \texttt{null} & \text{if } [\![\text{expr}]\!]_{G,u} = [w_1, \ldots, w_m], [\![\text{expr}']\!]_{G,u} = [w'_1, \ldots, w'_m] \\ & \text{with } \exists i, [\![w_i = w'_i]\!]_{G,u} = \texttt{null} \\ & \text{and } \forall i, [\![w_i = w'_i]\!]_{G,u} = \texttt{null} \text{ or } [\![w_i = w'_i]\!]_{G,u} = \texttt{true} \\ \texttt{false} & \text{otherwise} \end{cases}$

  where $w_1, \ldots, w_m, w'_1, \ldots, w'_m$ are values.

Paths. Assume that both expr and expr′ are expressions such that both $[\![\text{expr}]\!]_{G,u}$ and $[\![\text{expr}']\!]_{G,u}$ are path values.

- $[\![\text{expr} = \text{expr}']\!]_{G,u} = \begin{cases} \texttt{true} & \text{if } [\![\text{expr}]\!]_{G,u} = [\![\text{expr}']\!]_{G,u} \\ \texttt{false} & \text{otherwise} \end{cases}$

Mismatched composite types. If expr and expr′ are expressions such that $[\![\text{expr}]\!]_{G,u}$ is a value of a composite type (map, list, path) and $[\![\text{expr}']\!]_{G,u}$ is a non-null value of a different type, then $[\![\text{expr} = \text{expr}']\!]_{G,u} = \texttt{false}$. Conversely, if $[\![\text{expr}']\!]_{G,u}$ is of a composite type and $[\![\text{expr}]\!]_{G,u}$ is a non-null value of a different type, then $[\![\text{expr} = \text{expr}']\!]_{G,u} = \texttt{false}$.

Base types. If expr and expr′ are expressions such that $[\![\text{expr}]\!]_{G,u}$ and $[\![\text{expr}']\!]_{G,u}$ are non-null values of a non-composite type, then $[\![\text{expr} \star \text{expr}']\!]_{G,u}$ is allowed to be implementation-dependent, for $\star \in \{<, <=, >=, >, =, <>\}$. That is, for base types implementations have freedom when it comes to defining ordering. It is assumed however that for types considered here (numerical and strings), these are fixed and have their standard interpretation as ordering on numbers, and lexicographic ordering for strings.