Querying RDF Databases with Sub-CONSTRUCTs

Dominique Duval
LJK - Univ. Grenoble Alpes
dominique.duval@univ-grenoble-alpes.fr

Rachid Echahed
LIG - Univ. Grenoble Alpes
rachid.echahed@imag.fr

Frédéric Prost
LIG - Univ. Grenoble Alpes
frederic.prost@univ-grenoble-alpes.fr

Graph query languages feature mainly two kinds of queries when applied to a graph database: those inspired by relational databases which return tables such as SELECT queries and those which return graphs such as CONSTRUCT queries in SPARQL. The latter are object of study in the present paper. For this purpose, a core graph query language GrAL is defined with focus on CONSTRUCT queries. Queries in GrAL form the final step of a recursive process involving so-called GrAL patterns. By evaluating a query over a graph one gets a graph, while by evaluating a pattern over a graph one gets a set of matches which involves both a graph and a table. CONSTRUCT queries are based on CONSTRUCT patterns, and sub-CONSTRUCT patterns come for free from the recursive definition of patterns. The semantics of GrAL is based on RDF graphs with a slight modification which consists in accepting isolated nodes. Such an extension of RDF graphs eases the definition of the evaluation semantics, which is mainly captured by a unique operation called Merge. Besides, we define aggregations as part of GrAL expressions, which leads to an original local processing of aggregations.

1 Introduction

Graph database query languages are becoming ubiquitous. In contrast to classical relational databases where SQL language is a standard, different languages [1] have been proposed for querying graph databases, like SPARQL [9] or Cypher [5]. Among the most popular models for representing graph databases, one may quote for instance the *sets of triples* (or *RDF graphs* [10]) used by SPARQL or the *property graphs* used by Cypher. In addition to the lack of a standard model to represent graph databases, there are different kinds of queries in the context of graph query languages. One may essentially distinguish two classes of queries: those inspired by relational databases which return tables such as SELECT queries and those which return graphs such as CONSTRUCT queries in SPARQL. Such CONSTRUCT queries are graph-to-graph queries specific to graph databases.

The graph-to-graph queries received less attention than the graph-to-table queries. For instance, for the SPARQL language, a semantics of SELECT queries and subqueries is proposed in [6], a semantics of CONSTRUCT queries in [7] and a semantics of CONSTRUCT queries with nested CONSTRUCT queries in FROM clauses in [2, 8], where the outcome of a subCONSTRUCT is a graph. All these works consider graphs as sets of triples. Unfortunately, such a definition of graphs prevents having a uniform semantics of all patterns and in particular for BIND patterns.

In this paper, we focus on graph-to-graph queries and subqueries for RDF graphs and we propose a new semantics for CONSTRUCT subqueries which departs from the one in [2, 8]. First, we propose to change slightly the definition of graphs by allowing isolated nodes. Indeed, this new definition of graphs allows us to have a uniform semantics of all patterns. In fact, we define CONSTRUCT *subpatterns* rather

T. Kutsia (Ed.): Symbolic Computation
in Software Science (SCSS'21)
EPTCS 342, 2021, pp. 49–64, doi:10.4204/EPTCS.342.5

than CONSTRUCT subqueries. For this purpose, we introduce a core query language GrAL based on RDF graphs. The syntactic categories of GrAL include both queries and patterns. When evaluating a CONSTRUCT query over a graph one gets a graph, whereas when evaluating a CONSTRUCT pattern over a graph one gets a set of matches which involves both variable assignments and a graph. In fact, a CONSTRUCT query first acts as a CONSTRUCT pattern and then returns only the constructed graph. As the definition of patterns is recursive, CONSTRUCT subpatterns are obtained for free.
In order to define the semantics of GrAL, we introduce an algebra of operations over sets of matches, where a match is a morphism between graphs. We propose to base the semantics of GrAL upon an algebra on sets of matches, like the semantics of SQL is based upon relational algebra. All operations in our algebra essentially derive from a unique operation called *Merge*, which generalizes the well-known *Join* operation. As stated earlier, we consider graphs consisting of classical RDF triples possibly augmented with some additional isolated nodes. This slight extension helps formulating the semantics of patterns and queries without using some cumbersome notations to handle, for instance, environments defined by variable bindings. The proposed algebra is used to define an evaluation semantics for GrAL. As for aggregations, they are handled locally inside expressions. The semantics of the various patterns and queries is uniform, as it is based on instances of the *Merge* operation.
The paper is organized as follows. Section 2 introduces the algebra designed to express the semantics of the query language GrAL. In Section 3, the language GrAL is defined by its syntax and semantics. Concluding remarks are given in Section 4.

2 The Graph Query Algebra

The Graph Query Algebra is a family of operations which are used in Section 3 for defining the evaluation of queries in the Graph Algebraic Query Language GrAL. Graphs and matches are introduced in Section 2.1, then operations on sets of matches are defined in Section 2.2.

2.1 Graphs and matches

In this paper, graphs are kinds of generalised RDF graphs that may contain isolated nodes. Let $\mathscr{L}$ be a set, called the set of *labels*, union of two disjoint sets $\mathscr{C}$ and $\mathscr{V}$, called respectively the set of *constants* and the set of *variables*.

Definition 2.1 (graph)**.** Every element $t = (s, p, o)$ of $\mathscr{L}^3$ is called a *triple* and its members s, p and o are called respectively the *subject*, *predicate* and *object* of t. A *graph* X is made of a subset X_N of $\mathscr{L}$ called the set of *nodes* of X and a subset X_T of $\mathscr{L}^3$ called the set of *triples* of X, such that the subject and the object of each triple of X are nodes of X. The nodes of X which are neither a subject nor an object are called the *isolated nodes* of X. The set of *labels* of a graph X is the subset $\mathscr{L}(X)$ of $\mathscr{L}$ made of the nodes and predicates of X, then $\mathscr{C}(X) = \mathscr{C} \cap \mathscr{L}(X)$ and $\mathscr{V}(X) = \mathscr{V} \cap \mathscr{L}(X)$. Given two graphs X_1 and X_2, the graph X_1 is a *subgraph* of X_2, written $X_1 \subseteq X_2$, if $(X_1)_N \subseteq (X_2)_N$ and $(X_1)_T \subseteq (X_2)_T$, then obviously $\mathscr{L}(X_1) \subseteq \mathscr{L}(X_2)$. The *union* $X_1 \cup X_2$ is the graph defined by $(X_1 \cup X_2)_N = (X_1)_N \cup (X_2)_N$ and $(X_1 \cup X_2)_T = (X_1)_T \cup (X_2)_T$, then $\mathscr{L}(X_1 \cup X_2) = \mathscr{L}(X_1) \cup \mathscr{L}(X_2)$.

We will not use the *intersection* $X_1 \cap X_2$, which could be defined by $(X_1 \cap X_2)_N = (X_1)_N \cap (X_2)_N$ and $(X_1 \cap X_2)_T = (X_1)_T \cap (X_2)_T$: then the intersection of two graphs without isolated nodes might have isolated nodes and $\mathscr{L}(X_1 \cap X_2)$ might be strictly smaller than $\mathscr{L}(X_1) \cap \mathscr{L}(X_2)$, as for instance when $X_1 = \{(x,y,z)\}$ and $X_2 = \{(y,z,x)\}$ so that $X_1 \cap X_2 = \{x\}$.

Example 2.2. We introduce here a toy database representing a simplified view of a social media network. We will use it as a running example to illustrate various definitions. The network consists in *authors publishing messages*. Each message is *timestamped at* a certain *date* (a day). A message can *refer to* other messages and an author may *like* a message. An instance of such a network is described by the following graph G_0 (written "à la" RDF):

$G_0 =$
```
auth1 publishes mes1 .  auth1 publishes mes2 .
auth2 publishes mes3 .  auth3 publishes mes4 .  auth3 publishes mes5 .
mes1 stampedAt date1 .  mes2 stampedAt date2 .
mes3 stampedAt date1 .  mes4 stampedAt date4 .  mes5 stampedAt date4 .
mes3 refersTo mes1 .  mes4 refersTo mes1 .  mes4 refersTo mes2 .
auth1 likes mes3 .  auth1 likes mes4 .  auth1 likes mes5 .
auth2 likes mes1 .  auth2 likes mes4
```

The meaning of G_0 is that author `auth1` has published messages `mes1` and `mes2`, which have been stamped respectively at dates `date1` and `date2`, etc.

Definition 2.3 (match)**.** A *match* m from a graph X to a graph G, denoted $m : X \to G$, is a function from $\mathscr{L}(X)$ to $\mathscr{L}(G)$ which *preserves nodes* and *preserves triples* and which *fixes* $\mathscr{C}$, in the sense that $m(X_N) \subseteq G_N$, $m^3(X_T) \subseteq G_T$ and $m(x) = x$ for each x in $\mathscr{C}(X)$. The set of all matches from X to G is denoted $Match(X,G)$. An *isomorphism* of graphs is an invertible match.

When n is an isolated node of X then the node $m(n)$ does not have to be isolated in G. A match $m : X \to G$ determines two functions $m_N : X_N \to G_N$ and $m_T : X_T \to G_T$, restrictions of m and m^3 respectively. A match $m : X \to G$ is an isomorphism if and only if both functions $m_N : X_N \to G_N$ and $m_T : X_T \to G_T$ are bijections. This means that a function m from $\mathscr{L}(X)$ to $\mathscr{L}(G)$ is an isomorphism of graphs if and only if $\mathscr{C}(X) = \mathscr{C}(G)$ with $m(x) = x$ for each $x \in \mathscr{C}(X)$ and m is a bijection from $\mathscr{V}(X)$ to $\mathscr{V}(G)$: thus, X is the same as G up to variable renaming. It follows that the symbol used for naming a variable does not matter as long as graphs are considered only up to isomorphism.

Definition 2.4 (image of a graph by a function)**.** Let X be a graph. Every function $f : \mathscr{V}(X) \to \mathscr{L}$ can be extended in a unique way as a function $f' : \mathscr{L}(X) \to \mathscr{L}$ that fixes $\mathscr{C}$. The *image* $f(X)$ of X by f is the graph made of the nodes $f'(n)$ for $n \in X_N$ and the triples $(f')^3(t)$ for $t \in X_T$. The function f can be extended in a unique way as a match $f^\sharp : X \to f(X)$.

Definition 2.5 (compatible matches)**.** Two matches $m_1 : X_1 \to G_1$ and $m_2 : X_2 \to G_2$ are *compatible*, written as $m_1 \sim m_2$, if $m_1(x) = m_2(x)$ for each $x \in \mathscr{V}(X_1) \cap \mathscr{V}(X_2)$. Given two compatible matches $m_1 : X_1 \to G_1$ and $m_2 : X_2 \to G_2$, let $m_1 \bowtie m_2 : X_1 \cup X_2 \to G_1 \cup G_2$ denote the unique match such that $m_1 \bowtie m_2 \sim m_1$ and $m_1 \bowtie m_2 \sim m_2$ (which means that $m_1 \bowtie m_2$ coincides with m_1 on X_1 and with m_2 on X_2).

We will see in Section 3 that the execution of a query in GrAL is a graph-to-graph transformation, which main part is a graph-to-set-of-matches transformation.

Definition 2.6 (set of matches, assignment table)**.** Let X and G be graphs. A set $\underline{m}$ of matches, all of them from X to G, is denoted $\underline{m} : X \Rightarrow G$. The *assignment table* $Tab(\underline{m})$ of $\underline{m}$ is the two-dimensional table with the elements of $\mathscr{V}(X)$ in its first row, then one row for each m in $\underline{m}$, and the entry in row m and column x equals to $m(x)$.

Thus, the assignment table $Tab(\underline{m})$ describes the set of functions $\underline{m}|_{\mathscr{V}(X)} : \mathscr{V}(X) \Rightarrow \mathscr{L}$, made of the functions $m|_{\mathscr{V}(X)} : \mathscr{V}(X) \to \mathscr{L}$ for all $m \in \underline{m}$. The set of matches $\underline{m} : X \Rightarrow G$ is determined by the graphs X and G and the assignment table $Tab(\underline{m})$. This property is used hereafter to describe some examples.

Example 2.7. Here are some examples of matches in the graph G_0 (defined in Example 2.2).

- Let P_{ps} be the following graph (written "à la" SPARQL: variable names begin with "?"):

$$P_{ps} = \boxed{\texttt{?a publishes ?m . ?m stampedAt ?d}}$$

The set $\underline{m}_{ps}$ of all matches from P_{ps} to G_0 is:

$\underline{m}_{ps} : P_{ps} \Rightarrow G_0$ with $Tab(\underline{m}_{ps}) =$

?a	?m	?d
auth1	mes1	date1
auth1	mes2	date2
auth2	mes3	date1
auth3	mes4	date4
auth3	mes5	date4

- Let P_{pl} be the graph:

$$P_{pl} = \boxed{\texttt{?a1 publishes ?m . ?a2 likes ?m}}$$

The set $\underline{m}_{pl}$ of all matches from P_{pl} to G_0 is:

$\underline{m}_{pl} : P_{pl} \Rightarrow G_0$ with $Tab(\underline{m}_{pl}) =$

?a1	?m	?a2
auth1	mes1	auth2
auth2	mes3	auth1
auth3	mes4	auth1
auth3	mes4	auth2
auth3	mes5	auth1

- Let P'_{pl} be the following subgraph of P_{pl}, made of two isolated nodes:

$$P'_{pl} = \boxed{\texttt{?a1 . ?a2}}$$

The subset $\underline{m}'_{pl}$ of $\underline{m}_{pl}$ made of the restrictions to P'_{pl} of the matches in $\underline{m}_{pl}$ is:

$\underline{m}'_{pl} : P'_{pl} \Rightarrow G_0$ with $Tab(\underline{m}'_{pl}) =$

?a1	?a2
auth1	auth2
auth2	auth1
auth3	auth1
auth3	auth2

- Let P_{prp} be the graph:

$$P_{prp} = \boxed{\texttt{?a1 publishes ?m1 . ?m1 refersTo ?m2 . ?a2 publishes ?m2}}$$

The set $\underline{m}_{prp}$ of all matches from P_{prp} to G_0 is:

$\underline{m}_{prp} : P_{prp} \Rightarrow G_0$ with $Tab(\underline{m}_{prp}) =$

?a1	?m1	?m2	?a2
auth2	mes3	mes1	auth1
auth3	mes4	mes1	auth1
auth3	mes4	mes2	auth1

Definition 2.8 (image of a graph by a set of functions)**.** The *image* of a graph X by a set of functions $\underline{f}$ from $\mathscr{V}(X)$ to $\mathscr{L}$, denoted $\underline{f}(X)$, is the graph union of the graphs $f(X)$ for every f in $\underline{f}$. Every set of functions $\underline{f} : \mathscr{V}(X) \Rightarrow \mathscr{L}$ can be extended in a unique way as a set of matches $\underline{f}^\sharp : X \Rightarrow \underline{f}(X)$.

Remark 2.9 (about RDF graphs)**.** RDF graphs [10] are graphs (as in Definition 2.1) without isolated nodes, where constants are either IRIs (Internationalized Resource Identifiers) or literals and where all predicates are IRIs and only objects can be literals. Blank nodes in RDF graphs are the same as variable nodes in our graphs. Thus an isomorphism of RDF graphs, as defined in [10], is an isomorphism of graphs as in Definition 2.3.

2.2 Operations on sets of matches

In this Section we introduce some operations on sets of matches which are used in Section 3 for defining the semantics of GrAL. The prominent one is the *merging* operation (Definition 2.10), which is a kind of generalized *joining* operation (see Definition 2.14). Other basic operations are the simple *restriction* and *extension* operations (Definitions 2.12 and 2.13). Then, these basic operations are combined in order to get some derived operations (Definition 2.14).

Definition 2.10 (Merge)**.** Let $\underline{m} : X \Rightarrow G$ be a set of matches and $\underline{p}_m : Y \Rightarrow H_m$ a family of sets of matches indexed by $m \in \underline{m}$, and let $H = \cup_{m\in\underline{m}} H_m$. The *merging* of $\underline{m}$ along the family $(\underline{p}_m)_{m\in\underline{m}}$ is the set of matches $m \bowtie p$ for every $m \in \underline{m}$ and every $p \in \underline{p}_m$ compatible with m:
$Merge(\underline{m}, (\underline{p}_m)_{m\in\underline{m}}) = \{m \bowtie p \mid m \in \underline{m} \wedge p \in \underline{p}_m \wedge m \sim p\} : X \cup Y \Rightarrow G \cup H$.

Let $\underline{q} = Merge(\underline{m}, (\underline{p}_m)_{m\in\underline{m}})$, then $\underline{q}$ is made of a match $m \bowtie p$ for each pair (m, p) with $m \in \underline{m}$ and $p \in \underline{p}_m$ compatible with m (so that for each m in $\underline{m}$ the number of $m \bowtie p$ in $\underline{q}$ is between 0 and $Card(\underline{p}_m)$). The match $m \bowtie p : X \cup Y \to G \cup H$ is such that $m \bowtie p(x) = m(x)$ when $x \in X$ and $m \bowtie p(y) = p(y)$ when $y \in Y$, which is unambiguous because of the compatibility condition.

Example 2.11. Here are some examples of merging, based on the sets of matches in Example 2.7.

- Let $\underline{p}'_{pl}$ be similar to $\underline{m}'_{pl}$, up to renaming the variables `?a1` and `?a2` as `?a2` and `?a1`, respectively, so that $\underline{p}'_{pl}$ is:

$$\underline{p}'_{pl} : P'_{pl} \Rightarrow G_0 \text{ with } Tab(\underline{p}'_{pl}) =$$

?a1	?a2
auth2	auth1
auth1	auth2
auth1	auth3
auth2	auth3

  For each match m in $\underline{m}'_{pl}$ let $\underline{p}_m = \underline{p}'_{pl}$, which does not depend on m. Then $Merge(\underline{m}'_{pl}, (\underline{p}_m)_{m\in\underline{m}'_{pl}})$ is denoted simply $Join(\underline{m}'_{pl}, \underline{p}'_{pl})$ (as in Definition 2.14 and Remark 2.15) and:

$$\underline{q}'_{pl} = Join(\underline{m}'_{pl}, \underline{p}'_{pl}) : P'_{pl} \Rightarrow G_0 \text{ with } Tab(\underline{q}'_{pl}) =$$

?a1	?a2
auth1	auth2
auth2	auth1

- Assume that there is some operation *concat* that builds a string from any given date and string. For each match m in $\underline{m}_{ps}$ let $\underline{p}_m = \{p_m\} : \{?dm\} \Rightarrow H_{ps}$ where $p_m(?dm) = concat(m(?d), m(?m))$ and

H_{ps} is any graph that contains all strings $concat(?d,?m)$ as nodes. Then:

$$\underline{q}_{ps} = Merge(\underline{m}_{ps}, (\underline{p}_m)_{m\in\underline{m}_{ps}}) : P_{ps} \cup \{?dm\} \Rightarrow G_0 \cup H_{ps}$$

with $Tab(\underline{q}_{ps}) =$

?a	?m	?d	?dm
auth1	mes1	date1	date1mes1
auth1	mes2	date2	date2mes2
auth2	mes3	date1	date1mes3
auth3	mes4	date4	date4mes4
auth3	mes5	date4	date4mes5

- For each match m in $\underline{m}_{ps}$ let $\underline{p}_m = \{p_m\} : \{?r\} \Rightarrow H_{var}$ where $p_m(?r)$ is some fresh variable $?r_m$ and H_{var} is any graph that contains all variables $?r_m$ as nodes. Then:

$$\underline{q}_{var} = Merge(\underline{m}_{ps}, (\underline{p}_m)_{m\in\underline{m}_{ps}}) : P_{ps} \cup \{?r\} \Rightarrow G_0 \cup H_{var}$$

with $Tab(\underline{q}_{var}) =$

?a	?m	?d	?r
auth1	mes1	date1	?r1
auth1	mes2	date2	?r2
auth2	mes3	date1	?r3
auth3	mes4	date4	?r4
auth3	mes5	date4	?r5

Definition 2.12 (Restrict)**.** Let $\underline{m} : X \Rightarrow G$ be a set of matches. For every graph Y contained in X and every graph H contained in G such that $\underline{m}(Y) \subseteq H$, the *restriction* $Restrict(\underline{m},Y,H) : Y \Rightarrow H$ is made of the restrictions of the matches in $\underline{m}$ as matches from Y to H. When $H = G$ the notation may be simplified: $Restrict(\underline{m},Y) = Restrict(\underline{m},Y,G) : Y \Rightarrow G$.

Definition 2.13 (Extend)**.** Let $\underline{m} : X \Rightarrow G$ be a set of matches. For every graph H containing G, the *extension* $Extend(\underline{m},H) : X \Rightarrow H$ is made of the extensions of the matches in $\underline{m}$ as matches from X to H.

New operations are obtained by combining the previous ones (assuming that *true* is a constant). Comments on Definition 2.14 are given in Remark 2.15. We will see in Section 3.2 that these derived operations provide the semantics of the operators of the language GrAL.

Definition 2.14 (derived operations)**.**

- For every sets of matches $\underline{m} : X \Rightarrow G$ and $\underline{p} : Y \Rightarrow H$, let $\underline{p}_m = \underline{p}$ for each $m \in \underline{m}$, then:
  $Join(\underline{m},\underline{p}) = Merge(\underline{m},(\underline{p}_m)_{m\in\underline{m}}) : X \cup Y \Rightarrow G \cup H$.
- For every set of matches $\underline{m} : X \Rightarrow G$, every family of constants $\underline{c} = (c_m)_{m\in\underline{m}}$ and every variable x, let $\underline{p}_m = \{p_m\}$ and $p_m(x) = c_m$ for each $m \in \underline{m}$, then:
  $Bind(\underline{m},\underline{c},x) = Merge(\underline{m},(\underline{p}_m)_{m\in\underline{m}}) : X \cup \{x\} \Rightarrow G \cup \underline{c}$.
- For every set of matches $\underline{m} : X \Rightarrow G$ and every family of constants $\underline{c} = (c_m)_{m\in\underline{m}}$, for some fresh variable x, let $\underline{true} = (true)_{m\in\underline{m}}$:
  $Filter(\underline{m},\underline{c}) = Restrict(Bind(Bind(\underline{m},\underline{c},x),\underline{true},x),X,G) : X \Rightarrow G$.
- For every set of matches $\underline{m} : X \Rightarrow G$ and every graph R, for every $m \in \underline{m}$ let $p_m : R \to p_m(R)$ be the match such that:
  $p_m(x) = m(x)$ if $x \in \mathcal{V}(R) \cap \mathcal{V}(X)$
  and $p_m(x) = var(x,m)$ is a fresh variable if $x \in \mathcal{V}(R) \setminus \mathcal{V}(X)$
  and let $\underline{p}_m = \{p_m\}$ and $\underline{p}(R) = \cup_{m\in\underline{m}}(p_m(R))$, then:
  $Construct(\underline{m},R) = Restrict(Merge(\underline{m},(\underline{p}_m)_{m\in\underline{m}}),R) : R \Rightarrow G \cup \underline{p}(R)$.

- For every sets of matches $\underline{m} : X \Rightarrow G$ and $\underline{p} : X \Rightarrow H$:
  $Union(\underline{m}, \underline{p}) = Extend(\underline{m}, G \cup H) \cup Extend(\underline{p}, G \cup H) : X \Rightarrow G \cup H$.

Remark 2.15. Let us analyse these definitions. Note that the definition of *Bind* and *Filter* rely on the fact that isolated nodes are allowed in graphs.

- Operation *Join* is *Merge* when the set of matches $\underline{p}_m$ does not depend on m, so that:
  $Join(\underline{m}, \underline{p}) = \{m \bowtie p \mid m \in \underline{m} \wedge p \in \underline{p} \wedge m \sim p\} : X \cup Y \Rightarrow G \cup H$.
  It follows that *Join* is commutative.

- Operation *Bind* is *Merge* when $\underline{p}_m$ has exactly one element p_m for each m, which is such that $p_m(x) = c_m$. There are two cases:
  - If $x \in \mathscr{V}(X)$ then this operation selects the matches m in $\underline{m}$ such that $m(x) = c_m$:
    $Bind(\underline{m}, \underline{c}, x) = \{m \mid m \in \underline{m} \wedge m(x) = c_m\} : X \Rightarrow G$.
  - If $x \notin \mathscr{V}(X)$ then this operation extends each match m in $\underline{m}$ by assigning the value c_m to the variable x. Let us denote the resulting match as $m \uplus (x \mapsto c_m)$, so that:
    $Bind(\underline{m}, \underline{c}, x) = \{m \uplus (x \mapsto c_m) \mid m \in \underline{m}\} : X \cup \{x\} \Rightarrow G \cup \{\underline{c}\}$.

- Operation *Filter* applies *Bind* twice, first when $x \notin \mathscr{V}(X)$ for extending each $m \in \underline{m}$ by assigning c_m to x, then since $x \in \mathscr{V}(X \cup \{x\})$ for selecting the matches m in $\underline{m}$ such that $c_m = true$. Now the value of the auxiliary variable x is always *true*, so that x can be dropped: this is the role of the last step which restricts the domain of the matches from $X \cup \{x\}$ to X and its range from $G \cup \{\underline{c}\}$ to G.

- The first step in operation *Construct* is *Merge* when $\underline{p}_m$ has exactly one element p_m for each m (as for *Bind*), which is determined by $p_m(x) = var(x, m)$ for each variable x in R that does not occur in X. Each $var(x, m)$ is a fresh variable, which means that it is distinct from the variables in G, X and R, and that the variables $var(x, m)$ are pairwise distinct. Note that the precise symbol used for denoting $var(x, m)$ does not matter. The second step in operation *Construct* restricts the domain of the matches from $X \cup R$ to R. Thus:
  $Construct(\underline{m}, R)$ is the set of matches from R to $G \cup \underline{p}(R)$
  determined by the functions $f_m : \mathscr{V}(R) \to \mathscr{L}$ (for each $m \in \underline{m}$) such that
  $f_m(x) = m(x)$ if $x \in \mathscr{V}(R) \cap \mathscr{V}(X)$ and $f_m(x) = var(x, m)$ if $x \in \mathscr{V}(R) \setminus \mathscr{V}(X)$.
  Thus, the graph $G \cup \underline{p}(R)$ is obtained by "gluing" one copy of G with $Card(\underline{m})$ copies of R in the right way. Note that the functions f_m are pairwise distinct when $\mathscr{V}(R)$ is not included in $\mathscr{V}(X)$, but it needs not be the case in general. Also, note that the domain R of $Construct(\underline{m}, R)$ may be quite different from the domain X of $\underline{m}$, whereas every other operation in Definition 2.14 either keeps or extends the domain of $\underline{m}$.

- Operation *Union* is simply the set-theoretic union of sets of matches which share the same domain (by assumption) and the same range (by extending the range if necessary). This operation differs from the previous ones in the sense that it is not defined by examining the matches in its arguments. Note that *Union* is commutative.

Proposition 2.16. *The sets of matches obtained by the operations previously defined in this Section have bounded cardinals, as follows.*

$Card(Merge(\underline{m},(\underline{p}_m)_{m\in\underline{m}})) \leq \sum_{m\in\underline{m}}(Card(\underline{p}_m))$
$Card(Restrict(\underline{m},X,G)) \leq Card(\underline{m})$
$Card(Extend(\underline{m},H)) = Card(\underline{m})$
$Card(Join(\underline{m},\underline{p})) \leq Card(\underline{m}) \times Card(\underline{p})$
$Card(Bind(\underline{m},\underline{c},x)) = Card(\underline{m})$
$Card(Filter(\underline{m},\underline{c})) \leq Card(\underline{m})$
$Card(Construct(\underline{m},R)) \leq Card(\underline{m})$
$Card(Union(\underline{m},\underline{p})) \leq Card(\underline{m}) + Card(\underline{p})$

The proof of Proposition 2.16 follows easily from the definitions.

3 The Graph Algebraic Query Language

In this Section we introduce the syntax and semantics of the Graph Algebraic Query Language GrAL. There are three syntactic categories in GrAL: *expressions*, *patterns* and *queries*. Expressions are considered in Section 3.1. Patterns are defined in Section 3.2, their semantics is presented as an evaluation function which maps every pattern P and graph G to a set of matches $[[P]]_G$. Queries are defined in Section 3.3, they are essentially specific kinds of patterns and their semantics is easily derived from the semantics of patterns, the main difference is that the execution of a query on a graph returns simply a graph instead of a set of matches.
To each expression e or pattern P is associated a set of variables called its *in-scope variables* and denoted $\mathscr{V}(e)$ or $\mathscr{V}(P)$, respectively. An expression e is *over* a pattern P if $\mathscr{V}(e) \subseteq \mathscr{V}(P)$. In this Section, as in Section 2, the set of *labels* $\mathscr{L}$ is the union of the disjoint sets $\mathscr{C}$ and $\mathscr{V}$, of *constants* and *variables* respectively. We assume that the set $\mathscr{C}$ of constants contains the numbers and strings and the boolean values *true* and *false*, as well as a symbol $\perp$ for errors.

3.1 Expressions

The expressions of GrAL are built from the labels using operators, which are classified as either basic operators (unary or binary) and aggregation operators (always unary). Remember that typing constraints are not considered in this paper. Typically, and not exclusively, the sets Op_1, Op_2 and Agg of *basic unary* operators, *basic binary* operators and *aggregation* operators can be:
$Op_1 = \{-, \text{NOT}\}$,
$Op_2 = \{+,-,\times,/,=,>,<,\text{AND},\text{OR}\}$,
$Agg = Agg_{elem} \cup \{agg\ \text{DISTINCT} \mid agg \in Agg_{elem}\}$.
where $Agg_{elem} = \{\text{MAX},\text{MIN},\text{SUM},\text{AVG},\text{COUNT}\}$
A *group of expressions* is a non-empty finite list of expressions.

Definition 3.1 (syntax of expressions)**.** The *expressions* e of GrAL and their set of *in-scope* variables $\mathscr{V}(e)$ are defined recursively as follows:

- A constant $c \in \mathscr{C}$ is an expression with $\mathscr{V}(c) = \emptyset$.
- A variable $x \in \mathscr{V}$ is an expression with $\mathscr{V}(x) = \{x\}$.
- If e_1 is an expression and $op \in Op_1$ then $op\ e_1$ is an expression with $\mathscr{V}(op\ e_1) = \mathscr{V}(e_1)$.
- If e_1 and e_2 are expressions and $op \in Op_2$ then $e_1\ op\ e_2$ is an expression with $\mathscr{V}(e_1\ op\ e_2) = \mathscr{V}(e_1) \cup \mathscr{V}(e_2)$.

- If e_1 is an expression and $agg \in Agg$ then $agg(e_1)$ is an expression with $\mathcal{V}(agg(e_1)) = \mathcal{V}(e_1)$.
- If e_1 is an expression, $agg \in Agg$ and gp a group of expressions with all its variables distinct from the variables in e_1, then $agg(e_1 \text{ BY } gp)$ is an expression with $\mathcal{V}(agg(e_1 \text{ BY } gp)) = \mathcal{V}(e_1)$.

The *value* of an expression with respect to a set of matches $\underline{m}$ (Definition 3.2) is a family of constants $ev(\underline{m}, e) = (ev(\underline{m}, e)_m)_{m \in \underline{m}}$ indexed by the set $\underline{m}$. Each constant $ev(\underline{m}, e)_m$ depends on e and m and it may also depend on other matches in $\underline{m}$ when e involves aggregation operators. The *value* of a group of expressions $gp = (e_1, ..., e_k)$ with respect to $\underline{m}$ is the list $ev(\underline{m}, gp)_m = (ev(\underline{m}, e_1), ..., ev(\underline{m}, e_k))$. To each basic operator op is associated a function $[[op]]$ (or simply op) from constants to constants if op is unary and from pairs of constants to constants if op is binary. To each aggregation operator agg in Agg is associated a function $[[agg]]$ (or simply agg) from *multisets* of constants to constants. Note that each family of constants determines a multiset of constants: for instance a family $\underline{c} = (c_m)_{m \in \underline{m}}$ of constants indexed by the elements of a set of matches $\underline{m}$ determines the multiset of constants $\{|c_m \mid m \in \underline{m}|\}$, which is also denoted $\underline{c}$ when there is no ambiguity. Some aggregation operators agg in Agg_{elem} are such that $[[agg]](\underline{c})$ depends only on the set underlying the multiset $\underline{c}$, which means that $[[agg]](\underline{c})$ does not depend on the multiplicities in the multiset $\underline{c}$: for instance this is the case for MAX and MIN but not for SUM, AVG, COUNT. When $agg = agg_{elem}$ DISTINCT with agg_{elem} in Agg_{elem} then $[[agg]](\underline{c})$ is $[[agg_{elem}]]$ applied to the underlying set of $\underline{c}$. For instance, COUNT $(\underline{c})$ counts the number of elements of the multiset $\underline{c}$ with their multiplicies, while COUNT DISTINCT $(\underline{c})$ counts the number of distinct elements in $\underline{c}$.

Definition 3.2 (evaluation of expressions)**.** Let X be a graph, e an expression over X and $\underline{m} : X \Rightarrow Y$ a set of matches. The *value* of e with respect to $\underline{m}$ is the family of constants $ev(\underline{m}, e) = (ev(\underline{m}, e)_m)_{m \in \underline{m}}$ defined recursively as follows (with notations as in Definition 3.1):

- $ev(\underline{m}, c)_m = c$.
- $ev(\underline{m}, x)_m = m(x)$.
- $ev(\underline{m}, op\ e_1)_m = [[op]]\, ev(\underline{m}, e_1)_m$.
- $ev(\underline{m}, e_1\ op\ e_2)_m = ev(\underline{m}, e_1)_m\, [[op]]\, ev(\underline{m}, e_2)_m$.
- $ev(\underline{m}, agg(e_1))_m = [[agg]](ev(\underline{m}, e_1))$ (which is the same for every m in $\underline{m}$).
- $ev(\underline{m}, agg(e_1\ BY\ gp))_m = [[agg]](ev(\underline{m}|_{gp,m}, e_1))$ where $\underline{m}|_{gp,m}$ is the subset of $\underline{m}$ made of the matches m' in $\underline{m}$ such that $ev(\underline{m}, gp)_{m'} = ev(\underline{m}, gp)_m$ (which is the same for every m and m' in $\underline{m}$ such that $ev(\underline{m}, gp)_m = ev(\underline{m}, gp)_{m'}$).

Example 3.3. Let $\underline{m}_{pl}$ be as in Example 2.7. For every m in $\underline{m}_{pl}$ we have:

$$ev(\underline{m}_{pl}, \text{COUNT } (likes))_m = 5$$

whereas $ev(\underline{m}_{pl}, \text{COUNT } (likes \text{ BY } ?a1))_m$ depends on the match m in $\underline{m}_{pl}$:

$$ev(\underline{m}_{pl}, \text{COUNT } (likes \text{ BY } ?a1))_m = \begin{cases} 1 & \text{when } m(?a1) = \texttt{auth1} \\ 1 & \text{when } m(?a1) = \texttt{auth2} \\ 3 & \text{when } m(?a1) = \texttt{auth3} \end{cases}$$

Definition 3.4 (equivalence of expressions)**.** Two expressions over a graph X are *equivalent* if they have the same value with respect to every set of matches $\underline{m} : X \Rightarrow Y$.

3.2 Patterns

In Definition 3.5 the patterns of GrAL are built from graphs by using five operators: JOIN, BIND, FILTER, CONSTRUCT and UNION. In Definition 3.6 the semantics of patterns is given by an evaluation function. Some patterns have an associated graph called a *template*, such a pattern P may give rise to a query Q as explained in Section 3.3, then the result of query Q is built from the template of P.

Definition 3.5 (syntax of patterns)**.** The *patterns* P of GrAL, their set of *in-scope* variables $\mathscr{V}(P)$ and their *template* graph $\mathscr{T}(P)$ when it exists are defined recursively as follows.

- A graph is a pattern, called a *basic pattern*, and $\mathscr{V}(P)$ is the set of variables of the graph P.
- If P_1 and P_2 are patterns then P_1 JOIN P_2 is a pattern and $\mathscr{V}(P_1 \text{ JOIN } P_2) = \mathscr{V}(P_1) \cup \mathscr{V}(P_2)$.
- If P_1 is a pattern, e an expression over P_1 and x a variable then P_1 BIND e AS x is a pattern and $\mathscr{V}(P_1 \text{ BIND } e \text{ AS } x) = \mathscr{V}(P_1) \cup \{x\}$.
- If P_1 is a pattern and e an expression over P_1 then P_1 FILTER e is a pattern and $\mathscr{V}(P_1 \text{ FILTER } e) = \mathscr{V}(P_1)$.
- If P_1 is a pattern and R a graph then P_1 CONSTRUCT R,
  also written CONSTRUCT R WHERE P_1, is a pattern and $\mathscr{V}(P_1 \text{ CONSTRUCT } R) = \mathscr{V}(R)$.
  In addition this pattern has a template $\mathscr{T}(P_1 \text{ CONSTRUCT } R) = R$.
- If P_1 and P_2 are patterns with template and if $\mathscr{T}(P_1) = \mathscr{T}(P_2) = R$ then
  P_1 UNION P_2 is a pattern and $\mathscr{V}(P_1 \text{ UNION } P_2) = \mathscr{V}(R)$.
  In addition this pattern has a template $\mathscr{T}(P_1 \text{ UNION } P_2) = \mathscr{T}(P_1) = \mathscr{T}(P_2)$.

The *value* of a pattern over a graph is a set of matches, as defined now.

Definition 3.6 (evaluation of patterns)**.** The *value* of a pattern P of GrAL over a graph G is a set of matches $[[P]]_G : [P] \Rightarrow G^{(P)}$ from a graph $[P]$ that depends only on P to a graph $G^{(P)}$ that contains G. This value $[[P]]_G : [P] \Rightarrow G^{(P)}$ is defined inductively as follows (with notations as in Definition 3.1):

- If P is a basic pattern then $[[P]]_G = \mathit{Match}(P, G) : P \Rightarrow G$.
- $[[P_1 \text{ JOIN } P_2]]_G = \mathit{Join}([[P_1]]_G, [[P_2]]_{G^{(P_1)}}) : [P_1] \cup [P_2] \Rightarrow G^{(P_1)^{(P_2)}}$.
- $[[P_1 \text{ BIND } e \text{ AS } x]]_G = \mathit{Bind}([[P_1]]_G, \mathit{ev}([[P_1]]_G, e), x) : [P_1] \cup \{x\} \Rightarrow G^{(P_1)} \cup [[P_1]]_G(e)$.
- $[[P_1 \text{ FILTER } e]]_G = \mathit{Filter}([[P_1]]_G, \mathit{ev}([[P_1]]_G, e)) : [P_1] \Rightarrow G^{(P_1)}$.
- $[[P_1 \text{ CONSTRUCT } R]]_G = \mathit{Construct}([[P_1]]_G, R) : R \Rightarrow G^{(P_1)} \cup [[P_1]]_G(R)$.
- $[[P_1 \text{ UNION } P_2]]_G = \mathit{Union}([[P_1]]_G, [[P_2]]_{G^{(P_1)}}) : R \Rightarrow G^{(P_1)^{(P_2)}}$ where $R = \mathscr{T}(P_1) = \mathscr{T}(P_2)$.

Remark 3.7. Note that, syntactically, each operator OP builds a pattern P from a pattern P_1 and a parameter *param*, which is either a pattern P_2 (for JOIN and UNION), a pair (e, x) made of an expression and a variable (for BIND), an expression e (for FILTER) or a graph R (for CONSTRUCT). Semantically, for every non-basic pattern $P = P_1$ OP *param*, we denote $\underline{m}_1 : X_1 \Rightarrow G_1$ for $[[P_1]]_G : [P_1] \Rightarrow G^{(P_1)}$ and $\underline{m} : X \Rightarrow G'$ for $[[P]]_G : [P] \Rightarrow G^{(P)}$. In every case it is necessay to evaluate $\underline{m}_1$ before evaluating $\underline{m}$: for JOIN and UNION this is because pattern P_2 is evaluated on G_1, for BIND and FILTER because expression e is evaluated with respect to $\underline{m}_1$, and for CONSTRUCT because of the definition of *Construct*. According to Definition 3.6 given a pattern P and a graph G, the value $\underline{m} : X \Rightarrow G'$ of P is determined as follows:

- When P is a basic pattern then $X = P$, $G' = G$ and $\underline{m}$ is made of all matches from P to G.

- $P = P_1$ OP *param* then the semantics of P is easily derived from Definition 2.14 (see also Remark 2.15). However, note that the semantics of P_1 JOIN P_2 and P_1 UNION P_2 is not symmetric in P_1 and P_2 in general, unless $G^{(P_1)} = G$ and $G^{(P_2)} = G$, which occurs when P_1 and P_2 are basic patterns.
- The graph $G^{(P)}$ is built by adding to G "whatever is required" for the evaluation, in examples we often avoid its precise description.

Given a non-basic pattern $P = P_1$ OP *param*, the pattern P_1 is a *subpattern* of P, as well as P_2 when $P = P_1$ JOIN P_2 or $P = P_1$ UNION P_2. The semantics of patterns is defined in terms of the semantics of its subpatterns (and the semantics of its other arguments, if any). Thus, for instance, CONSTRUCT patterns can be nested at any depth.

Proposition 3.8. *For every pattern P, the set $\mathscr{V}(P)$ of in-scope variables of P is the same as the set $\mathscr{V}([P])$ of variables of the graph $[P]$.*

Example 3.9. In each item below we consider first some pattern P_i and some template R_i, then the pattern

$$C_i = P_i \text{ CONSTRUCT } R_i \text{ also written as } C_i = \text{CONSTRUCT } R_i \text{ WHERE } P_i \,.$$

We refer to Examples 2.7 and 3.3.

- ```
  C1 = CONSTRUCT { ?a1 cites ?a2 }
       WHERE { ?a1 publishes ?m1 . ?m1 refersTo ?m2 . ?a2 publishes ?m2 }
  ```
  Here, $C_1 = P_1$ CONSTRUCT R_1 where $P_1 = P_{prp}$, so that the value of P_1 over G_0 is $\underline{m}_{prp} : P_1 \Rightarrow G$. Note that $\mathscr{V}(R_1) \subseteq \mathscr{V}(P_1)$. Let $G_1 = G_0 \cup \{$`auth2 cites auth1 . auth3 cites auth1`$\}$, the value of C_1 over G_0 is:

$$[[C_1]]_{G_0} : R_1 \Rightarrow G_1 \text{ with } Tab([[C_1]]_{G_0}) =$$

?a1	?a2
auth2	auth1
auth3	auth1

- ```
  C2 = CONSTRUCT { ?n }
       WHERE {
         ?a likes ?m
         BIND COUNT(likes) AS ?n }
  ```
  Here, $C_2 = P_2$ CONSTRUCT R_2 where the template R_2 is the graph made of only one isolated node which is the variable $?n$. The graph $[P_2]$ is $\{$`?a likes ?m . ?n`$\}$. Let $G_2 = G_0 \cup \{5\}$, the value of C_2 over G_0 is:

$$[[P_2]]_{G_0} : [P_2] \Rightarrow G_2 \text{ with } Tab([[P_2]]_{G_0}) =$$

?a	?m	?n
auth1	mes3	5
auth1	mes4	5
auth1	mes5	5
auth2	mes1	5
auth2	mes4	5

  Then the graph $[C_2]$ is $\{$`?n`$\}$ and the value of C_2 over G_0 is:

$$[[C_2]]_{G_0} : [C_2] \Rightarrow G_2 \text{ with } Tab([[C_2]]_{G_0}) =$$

?n
5

- ```
  C3 = CONSTRUCT { ?a1 nbOfLikes ?n }
       WHERE { ?a1 publishes ?m . ?a2 likes ?m
         FILTER (NOT(?a1=?a2))
         BIND COUNT(likes BY ?a1) AS ?n }
  ```

  Here, $C_3 = P_3$ CONSTRUCT R_3 where $R_3 = \{$?a1 nbOfLikes ?n$\}$ is made of one triple and $[P_3] = \{$?a1 publishes ?m . ?a2 likes ?m . ?n$\}$. The evaluation of C_3 over G_0 starts from $\underline{m}_{pl}$. Let $G_3 = G_0 \cup \{$auth1 nbOfLikes 1 . auth2 nbOfLikes 1 . auth3 nbOfLikes 3$\}$, the value of C_3 over G_0 is:

$$[[P_3]]_{G_0} : [P_3] \Rightarrow G_3 \text{ with } Tab([[P_3]]_{G_0}) =$$

?a1	?m	?a2	?n
auth1	mes1	auth2	1
auth2	mes3	auth1	1
auth3	mes4	auth1	3
auth3	mes4	auth2	3
auth3	mes5	auth1	3

  Then the graph $[C_3]$ is $R_3 = \{$?a1 nbOfLikes ?n$\}$ and the value of C_3 over G_0 is:

$$[[C_3]]_{G_0} : [C_3] \Rightarrow G_3 \text{ with } Tab([[C_3]]_{G_0}) =$$

?a1	?n
auth1	1
auth2	1
auth3	3

- ```
  C4 = CONSTRUCT { ?a1 nbOfFriends ?n }
       WHERE
       {
         CONSTRUCT { ?a1 friend ?a2 }
         WHERE {
           ?a1 publishes ?m1 . ?a2 likes ?m1 .
           ?a2 publishes ?m2 . ?a1 likes ?m2
         }
         BIND COUNT (friend BY ?a1) AS ?n
       }
  ```

  Here $C_4 = P_4$ CONSTRUCT R_4 where P_4 itself contains a subpattern $C'_4 = P'_4$ CONSTRUCT R'_4. The evaluation of the basic pattern P'_4 over G_0 gives

$$[[P'_4]]_{G_0} : P'_4 \Rightarrow G_0 \text{ with } Tab([[P'_4]]_{G_0}) =$$

?a1	?m1	?a2	?m2
auth1	mes1	auth2	mes3
auth2	mes3	auth1	mes1

  Then we get $[C'_4] = \{$?a1 friend ?a2$\}$ and:

$$[[C'_4]]_{G_0} : [C'_4] \Rightarrow {G_0}^{(C'_4)} \text{ with } Tab([[C'_4]]_{G_0}) =$$

?a1	?a2
auth1	auth2
auth2	auth1

  Finally $[C_4] = \{$?a1 NbOfFriends ?n$\}$ and:

$$[[C_4]]_{G_0} : [C_4] \Rightarrow {G_0}^{(C_4)} \text{ with } Tab([[C_4]]_{G_0}) =$$

?a1	?n
auth1	1
auth2	1

- ```
  C5 = CONSTRUCT { ?r author ?a . ?r date ?d }
       WHERE { ?a publishes ?m . ?m stampedAt ?d }
  ```

  Here $C_5 = P_5$ CONSTRUCT R_5 with a variable $?r$ in R_5 that does not occur in P_5. The value of the basic pattern P_5 over G_0 is:

$$[[P_5]]_{G_0} : P_5 \Rightarrow G_0 \text{ with } Tab([[P_5]]_{G_0}) =$$

?a	?m	?d
auth1	mes1	date1
auth1	mes2	date2
auth2	mes3	date1
auth3	mes4	date4
auth3	mes5	date4

  Then $[C_5] = R_5$ and, since $[[P_5]]_{G_0}$ is made of 5 matches, the value $[[C_5]]_{G_0}$ is obtained by gluing 5 copies of R_5, with 5 fresh variables corresponding to different renamings of variable $?r$. Indeed, the variable $?r$ in R_5, which is not a variable of P_5, gives rise to one fresh variable for each match of P_5 in G_0. Thus:

$$[[C_5]]_{G_0} : R_5 \Rightarrow {G_0}^{(C_5)} \text{ with } Tab([[C_5]]_{G_0}) =$$

?r	?a	?d
?r1	auth1	date1
?r2	auth1	date2
?r3	auth2	date1
?r4	auth3	date4
?r5	auth3	date4

Definition 3.10 (equivalence of patterns)**.** Two patterns are *equivalent* if they have the same value over G for every graph G.

Proposition 3.11. *For every basic patterns P_1 and P_2, the basic pattern $P_1 \cup P_2$ is equivalent to P_1* JOIN *P_2 and to P_2* JOIN *P_1.*

3.3 Queries

A query in GrAL is essentially a pattern which has a template. The main difference between patterns and queries is that, while a pattern is interpreted as a function from graphs to sets of matches, a query is interpreted as a function from graphs to graphs. The operator for building queries from patterns is denoted GRAPH. According to Definition 3.6, the value of a pattern P with template R over a graph G is a set of matches $[[P]]_G : R \Rightarrow G^{(P)}$, and the semantics of patterns is defined recursively in terms of their values. Thus, patterns have a graph-to-set-of-matches semantics, while queries have a graph-to-graph semantics, as defined below, based on Definition 2.8 of the image of a graph by a set of functions.

Definition 3.12 (syntax of queries)**.** A *query* Q of GrAL is written GRAPH (P) where P is either a CONSTRUCT or a UNION pattern. Then *the pattern of* Q is P and the *template* $\mathscr{T}(Q)$ of Q is the template of P.

Definition 3.13 (result of queries)**.** The *result* of a query Q with pattern P and template R over a graph G is the subgraph of $G^{(P)}$ image of R by $[[P]]_G$, it is denoted $Result(Q,G)$.

Thus, when $Q =$ GRAPH $(P_1$ CONSTRUCT $R)$, the result of Q over G is the graph $Result(Q,G) = [[P_1]]_G(R)$ built by "gluing" the graphs $m(R)$ for $m \in [[P_1]]_G$, where $m(R)$ is a copy of R with each variable $x \in \mathscr{V}(R) \setminus \mathscr{V}(X)$ replaced by a fresh variable $var(x,m)$. And when $Q =$ GRAPH $(P_1$ UNION $P_2)$, the result of Q over G is the graph $Result(Q,G) = H_1 \cup H_2$ where $H_i = Result($GRAPH $(P_i), G)$ and the fresh variables occuring in H_1 are distinct from the ones in H_2.
  ```

Example 3.14. It is now easy to compute the result of the GrAL queries:

$$Q_i = \text{GRAPH } (C_i)$$

over G_0 when C_i is a pattern from Example 3.9. We know that the result of Q_i applied to G_0 is an instance of R_i when $\mathscr{V}(R_i) \subseteq \mathscr{V}(C_i)$, and that in general it is built by "gluing" together several instances of R_i (as for query Q_5 below).

- **Author citations.** Let us say that an author $a1$ *cites* an author $a2$ when $a1$ has published a message that refers to a message published by $a2$. In order to build the graph of author citations we use the query:
  $Q_1 = \text{GRAPH } (C_1)$.
  From $[[C_1]]_{G_0}$ we get the graph:

  $$\mathit{Result}(Q_1, G_0) = \{\ \texttt{auth2 cites auth1 . auth3 cites auth1}\ \}\ .$$

- **Number of likes.** Let us count the number of *likes* in the database. We can get this result by counting the number of triples with predicate `likes`, or equivalently the number of predicates `likes`. We use the query:
  $Q_2 = \text{GRAPH } (C_2)$.
  From $[[C_2]]_{G_0}$ we get the graph:

  $$\mathit{Result}(Q_2, G_0) = \{\ \texttt{5}\ \}$$

  This result is the number 5, which is considered in GrAL as a graph made of only one isolated node. Note that we would get a query equivalent to Q_2 by counting either the number of authors $?a$ who like a message (with multiplicity the number of messages liked by $?a$), or by counting the number of messages $?m$ which are liked by someone (with multiplicity the number of authors who like $?m$). This means that the line `BIND COUNT(likes) AS ?n` could be replaced either by `BIND COUNT(?a) AS ?n` or by `BIND COUNT(?m) AS ?n` .

- **Number of likes per author.** Let us now count the number of *likes per author* in the database, which means, for each author count the number of likes of messages published by this author, except for self-likes. We display the result as the graph made of the triples `?a1 nbOfLikes ?n` where $?n$ is the number of likes of author $?a1$, by using the query:
  $Q_3 = \text{GRAPH } (C_3)$.
  From $[[C_3]]_{G_0}$ we get the graph:

  $$\mathit{Result}(Q_3, G_0) = \{\ \texttt{auth1 nbOfLikes 1 . auth2 nbOfLikes 1 . auth3 nbOfLikes 3}\ \}\ .$$

- **Number of friends per author.** Let us count the number of friends of each author, where friendship is the symmetric relation between authors defined as follows: two authors are *friends* when each one likes a publication by the other (here self-friends are allowed). We use the query:
  $Q_4 = \text{GRAPH } (C_4)$.
  From $[[C_4]]_{G_0}$ we get the graph:

  $$\mathit{Result}(Q_4, G_0) = \{\ \texttt{auth1 nbOfFriends 1 . auth2 nbOfFriends 1}\ \}\ .$$

- **Generation of fresh variables.** Now let us build, for each author $?a$ and each message $?m$ published by $?a$ and stamped at date $?d$, a tree with a fresh variable as root and with two branches, one

named `author` towards ?a and the other one named `date` towards ?d. We use the query:
$Q_5 = \text{GRAPH}\ (C_5)$.
From $[[C_5]]_{G_0}$ we get the graph:

$$Result(Q_5, G_0) = T_1 \cup T_2 \cup T_3 \cup T_4 \cup T_5$$

where each T_i is the copy of R_5 corresponding to the i-th row in $Tab([[C_5]]_{G_0})$, so that:

$$\begin{aligned}
T_1 &= \{\ \texttt{?r1 author auth1 . ?r1 date date1}\ \} \\
T_2 &= \{\ \texttt{?r2 author auth1 . ?r2 date date2}\ \} \\
T_3 &= \{\ \texttt{?r3 author auth2 . ?r3 date date1}\ \} \\
T_4 &= \{\ \texttt{?r4 author auth3 . ?r4 date date4}\ \} \\
T_5 &= \{\ \texttt{?r5 author auth3 . ?r5 date date4}\ \}
\end{aligned}$$

In fact, query Q_5 "mimicks" the following SELECT query Q'_5:

```
SELECT ?a ?d
WHERE { ?a publishes ?m . ?m stampedAt ?d }
```

As explained in [4], the various copies of the variable ?r in the result of Q_5 act as identifiers for the rows in the table result of Q'_5 over G_0 (as for instance in SPARQL), which is obtained by dropping the column ?r from $Tab([[P_5]]_{G_0})$. Note that the table $Tab([[P_5]]_{G_0})$ has all its rows distinct by definition, whereas this becomes false when the column ?r is dropped.

Definition 3.15 (equivalence of queries)**.** Two queries are *equivalent* if they have the same template and the same result over every graph.

It follows that queries with equivalent patterns are equivalent, but this condition is not necessary.

Remark 3.16 (about SPARQL queries)**.** CONSTRUCT queries in SPARQL are similar to CONSTRUCT queries in GrAL: the variables in $\mathscr{V}(R) \setminus \mathscr{V}(X)$ in GrAL play the same role as the blank nodes in SPARQL. However the subCONSTRUCT patterns are specific to GrAL. There is no SELECT query in this core version of GrAL, however following [4] we may consider SELECT queries as kinds of CONSTRUCT queries.

4 Conclusion

We considered the problem of the evaluation of graph-to-graph queries, namely CONSTRUCT queries, possibly involving nested sub-queries. We proposed a new uniform evaluation semantics of such queries which rests on a recursive definition of the notion of patterns and a new definition of the considered graphs which are allowed to have isolated nodes. Hence, the evaluation of a pattern always yields a pair consisting of a graph and a set of matches (variable assignments). Notice that we did not tackle explicitly graph-to-table queries such as the well-known SELECT queries. We have shown recently in [4] that SELECT queries are particular case of CONSTRUCT queries. This stems from an easy encoding of tables as graphs. Thus, the proposed semantics can be extended immediately to SELECT queries involving Sub-SELECT queries.
The present work opens several perspectives including a generalization of the proposed semantics to other models of graphs such as *property graphs*. Such an extension needs to ensure the existence of the main operations of the proposed algebra such as the *Merge* operation. An operational semantics, based

on rewriting systems, which is faithful with the evaluation semantics proposed in this paper is under progress. Its underlying rewrite rules are inspired by the algebraic approach in [3].
Furthermore, the core language GrAL contains only simple patterns needed to illustrate our uniform semantics. Comparison with other expressions such as EXISTS(*pattern*) or patterns such as FROM(*query*) [2, 8] remains to be investigated.

References

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter & Domagoj Vrgoc (2017): *Foundations of Modern Query Languages for Graph Databases*. *ACM Comput. Surv.* 50(5), pp. 68:1–68:40, doi:10.1145/3104031.

[2] Renzo Angles & Claudio Gutiérrez (2011): *Subqueries in SPARQL*. In Pablo Barceló & Val Tannen, editors: *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011*, *CEUR Workshop Proceedings* 749, CEUR-WS.org. Available at http://ceur-ws.org/Vol-749/paper19.pdf.

[3] Dominique Duval, Rachid Echahed & Frédéric Prost (2020): *An Algebraic Graph Transformation Approach for RDF and SPARQL*. In Berthold Hoffmann & Mark Minas, editors: *Proceedings of the Eleventh International Workshop on Graph Computation Models, GCM@STAF 2020, Online-Workshop, 24th June 2020*, *EPTCS* 330, pp. 55–70, doi:10.4204/EPTCS.330.4.

[4] Dominique Duval, Rachid Echahed & Frédéric Prost (2020): *All You Need Is CONSTRUCT*. *CoRR* abs/2010.00843. Available at https://arxiv.org/abs/2010.00843.

[5] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer & Andrés Taylor (2018): *Cypher: An Evolving Query Language for Property Graphs*. In Gautam Das, Christopher M. Jermaine & Philip A. Bernstein, editors: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, ACM, pp. 1433–1445, doi:10.1145/3183713.3190657.

[6] Mark Kaminski, Egor V. Kostylev & Bernardo Cuenca Grau (2017): *Query Nesting, Assignment, and Aggregation in SPARQL 1.1*. *ACM Trans. Database Syst.* 42(3), pp. 17:1–17:46, doi:10.1145/3083898.

[7] Egor V. Kostylev, Juan L. Reutter & Martín Ugarte (2015): *CONSTRUCT Queries in SPARQL*. In Marcelo Arenas & Martín Ugarte, editors: *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*, *LIPIcs* 31, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 212–229, doi:10.4230/LIPIcs.ICDT.2015.212.

[8] Axel Polleres, Juan L. Reutter & Egor V. Kostylev (2016): *Nested Constructs vs. Sub-Selects in SPARQL*. In Reinhard Pichler & Altigran Soares da Silva, editors: *Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016*, *CEUR Workshop Proceedings* 1644, CEUR-WS.org. Available at http://ceur-ws.org/Vol-1644/paper10.pdf.

[9] (2013): *SPARQL 1.1 Query Language*. W3C Recommendation. Available at https://www.w3.org/TR/sparql11-query/.

[10] (2014): *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. Available at https://www.w3.org/TR/rdf11-concepts/.