A Rule-Based Procedure for Graph Query Solving

Dominique Duval, Rachid Echahed, Frédéric Prost

A Rule-Based Procedure for Graph Query Solving*

Dominique Duval, Rachid Echahed, and Frédéric Prost

University Grenoble Alpes, Grenoble, France
firstname.lastname@imag.fr

Abstract. We consider a core language for graph queries. These queries, which may transform graphs to graphs, are seen as formulas to be solved with respect to graph databases. For this purpose, we first define a graph query algebra where some operations over graphs and sets of graph homomorphisms are specified. Then, the notion of pattern is introduced to represent a kind of recursively defined formula over graphs. The syntax and formal semantics of patterns are provided. Afterwards, we propose a new sound and complete procedure to solve patterns. This procedure, which is based on a set of rewriting rules, is terminating and develops only one needed derivation per pattern to be solved. Our procedure is generic in the sense that it can be adapted to different kinds of graph queries provided that the notions of graph and graph homomorphism are well defined.

Keywords: Rewriting systems, Graph Query Solving, Graph Databases

1 Introduction

Current developments in database theory show a clear shift from relational to graph databases [35]. Relational databases are now well mastered and have been largely investigated in the literature with an ISO standard language SQL [12, 15]. On the other side, graphs are being widely used as a flexible data model for numerous database applications [35]. So that various graph query languages such as SPARQL [37], Cypher [23] or G-CORE [2] to quote a few, as well as an ongoing ISO project of a standard language, called GQL[1], have emerged for graph databases.

Representing data graphically is quite legible. However, there is always a dilemma in choosing the right notion of graphs when modeling applications. This issue is already present in some well investigated domains such as modeling languages [8] or graph transformation [34]. Graph data representation does not escape from such dilemma. We can quote for example RDF graphs [38] on which SPARQL is based or variants of Property Graphs [23] currently used in several languages such as Cypher, G-CORE or in GQL.

* This work has been partly funded by the project VERIGRAPH : ANR-21-CE48-0015

[1] https://www.gqlstandards.org/

In addition to the possibility of using different graph representations for data, graph database languages feature new kinds of queries such as graph-to-graph queries, cf. CONSTRUCT queries in SPARQL or G-CORE, besides the classical graph-to-relation (table) queries such as SELECT or MATCH queries in SPARQL or Cypher. The former constitute a class of queries which transform a graph database into another graph database. The latter transform a graph into a multiset of solutions represented in general by means of a table just as in the classical relational framework.

In general, graph query processing integrates features shared with graph transformation techniques and goal solving or logic programming (variable assignments). Our main aim in this paper is to define an operational semantics, based on rewriting techniques, for graph queries. We propose a generic rule-based calculus, called *gq-narrowing* which is parameterized by the actual interpretations of graphs and their matches (homomorphisms). That is to say, the obtained calculus can be adapted to different definitions of graphs and the corresponding notion of match. The proposed calculus consists of a dedicated rewriting system and a narrowing-like [4, 25, 25] procedure which follows closely the formal semantics of patterns or queries, the same way as (SLD-)Resolution calculus is related to formal models underlying Horn or Datalog [27] clauses. The use of rewriting techniques in defining the proposed operational semantics paves the way to syntactic analysis and automated verification techniques for the proposed core language.

In order to define a sound and complete calculus, we first propose a uniform formal semantics for queries. For practical reasons, we were inspired by existing graph query languages and consider graph-to-graph queries and graph-to-table queries as two facets of one same syntactic object that we call *pattern*. The proposed patterns can be nested at will as in declarative functional terms and may include aggregation operators as well as graph construction primitives. The semantics of a pattern is defined as a set of matches, that is to say, a set of graph homomorphisms and not only a set of variable assignments as proposed in [3, 23, 22]. From such a set of matches, one can easily display either the table by considering the images of the variables as defined by the matches or the graph, target of the matches, or even both the table and the graph. The proposed semantics for patterns allows also to write nested patterns in a natural way, that is, new data graphs can be constructed on the fly before being queried.

The paper is organized as follows: The next section introduces a graph query algebra featuring some key operations needed to express the proposed calculus. Section 3 defines the syntax of patterns and queries as well as their formal semantics. In Section 4, a sound and complete calculus is given. First we introduce a rewriting system describing how query results are found. Then, we define gq-narrowing, which is associated with the proposed rules. Concluding remarks and related work are given in Section 5. Due to lack of space, omitted proofs as well as a new query form combining CONSTRUCT and SELECT query forms can be found in [20].

2 A Graph Query Algebra

During graph query processing, different intermediate results can be computed and composed. In this section, we introduce a Graph Query Algebra $\mathcal{GQ}$ which consists of a family of operations over graphs, matches (graph homomorphisms) and expressions. These different items are used later on to define the semantics of queries in Sections 3 and 4.

The algebra $\mathcal{GQ}$ is defined over a signature Σ_{gq}. The main sorts of Σ_{gq} are Gr, Som, Exp and Var to be interpreted as graphs, sets of matches, expressions and variables, respectively. The sort Var is a subsort of Exp. The main operators of Σ_{gq} are:

$\text{Match} : \text{Gr}, \text{Gr} \to \text{Som}$	$\text{Join} : \text{Som}, \text{Som} \to \text{Som}$
$\text{Bind} : \text{Som}, \text{Exp}, \text{Var} \to \text{Som}$	$\text{Filter} : \text{Som}, \text{Exp} \to \text{Som}$
$\text{Build} : \text{Som}, \text{Gr} \to \text{Som}$	$\text{Union} : \text{Som}, \text{Som} \to \text{Som}$

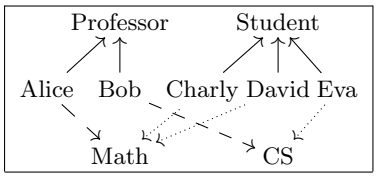
The above sorts and operators are given as an indication while being inspired by concrete languages. They may be modified or tuned according to actual graph query languages. Various interpretations of sorts Gr and Som can be given. In order to provide concrete examples, we have to fix an actual interpretation of these sorts. For all the examples given in the paper, we have chosen to interpret the sort Gr as generalized RDF graphs [38]. This choice is not a limitation, we might have chosen other notions of graphs such as property graphs [23]. Our choice here is motivated by the simplicity of the RDF graph definition (set of triples). Below, we define generalized RDF graphs. They are the usual RDF graphs with the ability to contain isolated nodes. Let $\mathcal{L}$ be a set, called the set of *labels*, made of the union of two disjoint sets $\mathcal{C}$ and $\mathcal{V}$, called respectively the set of *constants* and the set of *variables*.

Definition 1 (graph). *Every element $t = (s, p, o)$ of $\mathcal{L}^3$ is called a* triple *and its members s, p and o are called respectively the* subject*, the* predicate *and the* object *of t. A* graph *G is a pair $G = (G_N, G_T)$ made of a finite subset G_N of $\mathcal{L}$ called the set of* nodes *of G and a finite subset G_T of $\mathcal{L}^3$ called the set of* triples *of G, such that the subject and the object of each triple of G are nodes of G. The nodes of G which are neither a subject nor an object are called the* isolated nodes *of G. The set of* labels *of a graph G is the subset $\mathcal{L}(G)$ of $\mathcal{L}$ made of the nodes and predicates of G, then $\mathcal{C}(G) = \mathcal{C} \cap \mathcal{L}(G)$ and $\mathcal{V}(G) = \mathcal{V} \cap \mathcal{L}(G)$. The graph with an empty set of nodes and an empty set of triples is called the* empty graph *and is denoted by $\emptyset$. Given two graphs G_1 and G_2, the graph G_1 is a* subgraph *of G_2, written $G_1 \subseteq G_2$, if $(G_1)_N \subseteq (G_2)_N$ and $(G_1)_T \subseteq (G_2)_T$, thus $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$. The* union *$G_1 \cup G_2$ is the graph defined by $(G_1 \cup G_2)_N = (G_1)_N \cup (G_2)_N$ and $(G_1 \cup G_2)_T = (G_1)_T \cup (G_2)_T$, then $\mathcal{L}(G_1 \cup G_2) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$.*

In the rest of the paper we write graphs as sets of triples and nodes: for example $G = \{(s_1, o_1, p_1), n_1, n_2\}$ is the graph with four nodes n_1, n_2, s_1, p_1 and one triple (s_1, o_1, p_1).

Example 1. We define a toy database which is used as a running example throughout the paper. The database consists of *persons* who *are* either *professors* or *students*, with *topics* such that each professor (resp. student) *teaches* (resp. *studies*) some topics. The graph G_{ex} is described below by its triples (on the left) and by a diagram (on the right), in which plain arrows represent *is* relation, dashed lines represent *teaches* relation and dotted lines represent *studies* relation.

(Alice, is, Professor)	(Alice, teaches, Math)
(Bob, is, Professor)	(Bob, teaches, CS)
(Charly, is, Student)	(Charly, studies, Math)
(David, is, Student)	(David, studies, Math)
(Eva, is, Student)	(Eva, studies, CS)



Definition 2 (match). *A* graph homomorphism *from a graph L to a graph G, denoted $m : L \to G$, is a function from $\mathcal{L}(L)$ to $\mathcal{L}(G)$ which* preserves nodes *and* preserves triples*, in the sense that $m(L_N) \subseteq G_N$ and $m^3(L_T) \subseteq G_T$. A* match *is a graph homomorphism $m : L \to G$ which* fixes $\mathcal{C}$*, in the sense that $m(c) = c$ for each c in $\mathcal{C}(L)$.*

A match $m : L \to G$ determines two functions $m_N : L_N \to G_N$ and $m_T : L_T \to G_T$, restrictions of m and m^3 respectively. A match $m : L \to G$ is invertible if and only if both functions m_N and m_T are bijections. This means that a function m from $\mathcal{L}(L)$ to $\mathcal{L}(G)$ is an invertible match if and only if $\mathcal{C}(L) = \mathcal{C}(G)$ with $m(c) = c$ for each $c \in \mathcal{C}(L)$ and m is a bijection from $\mathcal{V}(L)$ to $\mathcal{V}(G)$: thus, L 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 invertible matches.

Notice that RDF graphs [38] are graphs according to Definition 1 but without isolated nodes, and 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. An isomorphism of RDF graphs, as defined in [38], is an invertible match.

Below we introduce some useful definitions on matches. Notice that we do not consider a match m as a simple variable assignment but rather as a graph homomorphism with clear source and target graphs. This nuance in the definition of matches is key in the rest of the paper since it allows us to define the notion of nested patterns in a straightforward manner.

Definition 3 (compatible matches). *Two matches $m_1 : L_1 \to G_1$ and $m_2 : L_2 \to G_2$ are* compatible*, written as $m_1 \sim m_2$, if $m_1(x) = m_2(x)$ for each $x \in \mathcal{V}(L_1) \cap \mathcal{V}(L_2)$. Given two compatible matches $m_1 : L_1 \to G_1$ and $m_2 : L_2 \to G_2$, let $m_1 \bowtie m_2 : L_1 \cup L_2 \to G_1 \cup G_2$ 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 L_1 and with m_2 on L_2).*

Definition 4 (building a match). *Let $m : L \to G$ be a match and R a graph. The match Build-match $(m, R) : R \to G \cup H_{m,R}$ is the unique match (up to*

variable renaming) such that for each variable x in R:

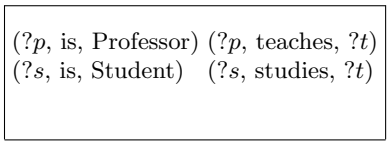
$$\textit{Build-match}\,(m,R)(x) = \begin{cases} m(x) \text{ if } x \in \mathcal{V}(R) \cap \mathcal{V}(L), \\ \textit{some fresh variable } var(m,x) \text{ if } x \in \mathcal{V}(R) - \mathcal{V}(L) \end{cases}$$

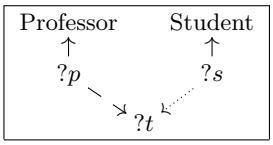
and $H_{m,R}$ is the image of R by Build-match (m,R).

Definition 5 (set of matches, assignment table). *Let L and G be graphs. A set $\underline{m}$ of matches, all of them from L to G, is denoted $\underline{m} : L \Rightarrow G$ and called a* homogeneous set of matches*, or simply a* set of matches*, with* source L *and* target G*. The* image *of L by $\underline{m}$ is the subgraph $\underline{m}(L) = \cup_{m \in \underline{m}}(m(L))$ of G. We denote $Match(L,G) : L \Rightarrow G$ the set of all matches from L to G. When L is the empty graph this set has one element which is the inclusion $\emptyset_G : \emptyset \to G$. We denote $\underline{i}_G = Match(\emptyset, G) : \emptyset \Rightarrow G$ this singleton and $\varnothing_G : \emptyset \Rightarrow G$ its empty subset. The* assignment table *$Tab(\underline{m})$ of $\underline{m}$ is the two-dimensional table with the elements of $\mathcal{V}(L)$ 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 $\underline{m}|_{\mathcal{V}(L)} : \mathcal{V}(L) \Rightarrow \mathcal{L}$, made of the functions $m|_{\mathcal{V}(L)} : \mathcal{V}(L) \to \mathcal{L}$ for all $m \in \underline{m}$. A set of matches $\underline{m} : L \Rightarrow G$ is determined by the graphs L and G and the assignment table $Tab(\underline{m})$.

Example 2. In order to determine whether professor $?p$ teaches topic $?t$ which is studied by student $?s$, we may consider the following graph L_{ex}, where $?p$, $?t$ and $?s$ are variables. In all examples, variables are preceded by a "?".





There are three matches from L_{ex} to the graph G_{ex} given in Example 1. The set $\underline{m}_{ex}$ of all these matches is $\underline{m}_{ex} = \{m_1, m_2, m_3\} : L_{ex} \Rightarrow G_{ex}$:

$\underline{m}_{ex} : L_{ex} \Rightarrow G_{ex}$ with $Tab(\underline{m}_{ex}) =$

$\underline{m}_{ex}$	$?p$	$?t$	$?s$
m_1	Alice	Maths	Charly
m_2	Alice	Maths	David
m_3	Bob	CS	Eva

Query languages usually provide a term algebra dedicated to express operations over integers, booleans and so forth. We do not care here about the way basic operations are chosen but we want to deal with aggregation operations as in most database query languages. Thus, one can think of any kind of term algebra with operators which are classified as either basic operators (unary or binary) and aggregation operators (always unary). For defining the syntax and semantics of aggregation functions we follow [19]. We consider that all expressions are well

typed. Typically, and not exclusively, the sets Op_1, Op_2 and Agg of *basic unary* operators, *basic binary* operators and *aggregation* operators can be:
$Op_1 = \{-, \text{not}\}$,
$Op_2 = \{+, -, \times, /, =, >, <, \text{and}, \text{or}\}$,
$Agg = \{\max, \min, \text{sum}, \text{avg}, \text{count}\}$.

Definition 6 (syntax of expressions). Expressions e *and their sets of* in-scope *variables* $\mathcal{V}(e)$ *are defined recursively as follows, with* $c \in \mathcal{C}$, $x \in \mathcal{V}$, $op_1 \in Op_1$, $op_2 \in Op_2$, $agg \in Agg$ *and* g *a set of expressions:*
$e ::= c \mid x \mid op_1\ e \mid e\ op_2\ e \mid agg(e) \mid agg(e \text{ by } g)$,
$\mathcal{V}(c) = \varnothing$, $\mathcal{V}(x) = \{x\}$, $\mathcal{V}(op_1\ e) = \mathcal{V}(e)$, $\mathcal{V}(e\ op_2\ e') = \mathcal{V}(e) \cup \mathcal{V}(e')$,
$\mathcal{V}(agg(e)) = \mathcal{V}(e)$ *and* $\mathcal{V}(agg(e \text{ by } g)) = \mathcal{V}(e)$
(variables in g *must be distinct from those in* e*).*

The *value* of an expression e with respect to a set of matches $\underline{m}$, as stated in Definition 7, is a family of constants $\underline{m}(e) = (m(e)_{\underline{m}})_{m \in \underline{m}}$ indexed by the set $\underline{m}$. In general $m(e)_{\underline{m}}$ depends on e and m and it may also depend on other matches in $\underline{m}$ when e involves aggregation operators. Whenever e is free from any aggregation operator then $m(e)_{\underline{m}}$ does not depend on the matches different from m in $\underline{m}$, so that it can be written simply $m(e)$. 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 is associated a function $[[agg]]$ (or simply agg) from *multisets* of constants to constants. Note that each family of constants $\underline{c} = (c_m)_{m \in \underline{m}}$ determines a multiset of constants $\{\!|c_m \mid m \in \underline{m}|\!\}$, which is also denoted $\underline{c}$ when there is no ambiguity.

Definition 7 (evaluation of expressions). *Let* L, G *be graphs,* e *an expression such that* $\mathcal{V}(e) \subseteq \mathcal{V}(L)$ *and* $\underline{m} : L \Rightarrow G$ *a set of matches. The* value *of* e *with respect to* $\underline{m}$ *is the family*
$\underline{m}(e) = (m(e)_{\underline{m}})_{m \in \underline{m}}$
defined recursively as follows. It is assumed that each $m(e)_{\underline{m}}$ *in this definition is a constant.*
$m(c)_{\underline{m}} = c$, $m(x)_{\underline{m}} = m(x)$, $m(op_1\ e)_{\underline{m}} = [[op_1]]\, m(e)_{\underline{m}}$,
$m(e\ op_2\ e')_{\underline{m}} = m(e)_{\underline{m}}\, [[op_2]]\, m(e')_{\underline{m}}$, $m(agg(e))_{\underline{m}} = [[agg]](\underline{m}(e))$,
$m(agg(e \text{ by } g))_{\underline{m}} = [[agg]](\underline{m}|_{g,m}(e))$ *where* $\underline{m}|_{g,m}$ *is*
the group of m in $\underline{m}$ with respect to g, *i.e., the subset of* $\underline{m}$ *made of the matches* m' *in* $\underline{m}$ *such that* $m'(e')_{\underline{m}} = m(e')_{\underline{m}}$ *for every expression* e' *in* g.
Note that $m(agg(e))_{\underline{m}}$ *is the same for all* m *in* $\underline{m}$*, while* $m(agg(e \text{ by } g))_{\underline{m}}$ *is the same for all* m *in* $\underline{m}$ *which are in a common group with respect to* g.

Example 3. Consider $\underline{m}_{ex} = \{m_1, m_2, m_3\} : L_{ex} \Rightarrow G_{ex}$ as in Example 2, denoted simply $\underline{m}$ for readability. Let us evaluate the expressions count($?s$) and count($?s$ by $?p$).
The evaluation of count($?s$) with respect to $\underline{m}$ runs as follows:

$\underline{m}(\text{count}(?s)) = (\, m_i(\text{count}(?s))_{\underline{m}} \,)_{i=1,2,3}$
$m_i(\text{count}(?s))_{\underline{m}} = \text{count}(\underline{m}(?s))$
$\underline{m}(?s) = (\, m_i(?s)_{\underline{m}} \,)_{i=1,2,3}$
$m_i(?s)_{\underline{m}} = m_i(?s) \quad \text{for } i = 1, 2, 3$
Since $m_1(?s) = \text{Charly}$, $m_2(?s) = \text{David}$ and $m_3(?s) = \text{Eva}$ we get:
$\underline{m}(?s) = (\, \text{Charly}, \text{David}, \text{Eva} \,)$
$\text{count}(\underline{m}(?s)) = 3$
$m_i(\text{count}(?s))_{\underline{m}} = 3 \quad \text{for } i = 1, 2, 3$
$\underline{m}(\text{count}(?s)) = (\, 3, 3, 3 \,)$
The evaluation of $\text{count}(?s \text{ by } ?p)$ with respect to $\underline{m}$ runs as follows:
$\underline{m}(\text{count}(?s \text{ by } ?p)) = (\, m_i(\text{count}(?s \text{ by } ?p))_{\underline{m}} \,)_{i=1,2,3}$
$m_i(\text{count}(?s \text{ by } ?p))_{\underline{m}} = \text{count}(\underline{m}|_{\{?p\},m_i}(?s))$
Since $m_1(?p) = m_2(?p) = \text{Alice}$ and $m_3(?p) = \text{Bob}$ we get
$\underline{m}|_{\{?p\},m_1} = \underline{m}|_{\{?p\},m_2} = \{m_1, m_2\}$ and $\underline{m}|_{\{?p\},m_3} = \{m_3\}$
Then $\text{count}(\underline{m}|_{\{?p\},m_i}(?s)) = 2$ for $i = 1, 2$ and $\text{count}(\underline{m}|_{\{?p\},m_3}(?s)) = 1$
and finally $\underline{m}(\text{count}(?s \text{ by } ?p)) = (\, 2, 2, 1 \,)$.

We conclude this section with the definition of the algebra $\mathcal{GQ}$ over Σ_{gq}. Whenever needed, we extend the target of matches: for every graph H and every match $m : L \to G$ where G is a subgraph of H we write $m : L \to H$ when m is considered as a match from L to H.

Definition 8 ($\mathcal{GQ}$ algebra). *The algebra $\mathcal{GQ}$ is the algebra over the signature Σ_{gq} where the sorts* Gr, Som, Exp *and* Var *are interpreted respectively as the set of graphs (Definition 1), the set of sets of matches (Definition 5), the set of expressions (Definition 6) and its subset of variables, and where the operators are interpreted by the operations with the same name, as follows.*

- *For all graphs L and G:*
  $Match(L, G) : L \Rightarrow G$ is the set of all matches from L to G.
- *For all sets of matches $\underline{m} : L \Rightarrow G$ and $\underline{p} : R \Rightarrow H$:*
  $Join(\underline{m}, \underline{p}) = \{m \bowtie p \mid m \in \underline{m} \wedge p \in \underline{p} \wedge m \sim p\} : L \cup R \Rightarrow G \cup H$.
- *For each set of matches $\underline{m} : L \Rightarrow G$ and each expression e, let $H_m = \{m(e)_{\underline{m}} \mid m \in \underline{m}\}$ and for each match m in $\underline{m}$ let $p_m : \{x\} \Rightarrow H_m$ be the match such that $p_m(x) = m(e)_{\underline{m}}$. Then for each fresh variable $x \notin \mathcal{V}(L)$:*
  $Bind(\underline{m}, e, x) = \{m \bowtie p_m \mid m \in \underline{m}\} : L \cup \{x\} \Rightarrow G \cup H_m$.
- *For each set of matches $\underline{m} : L \Rightarrow G$ and each expression e:*
  $Filter(\underline{m}, e) = \{m \mid m \in \underline{m} \wedge m(e)_{\underline{m}} = true\} : L \Rightarrow G$.
- *For each set of matches $\underline{m} : L \Rightarrow G$ and each graph R:*
  $Build(\underline{m}, R) = \{Build\text{-}match\,(m, R) \mid m \in \underline{m}\} : R \Rightarrow G \cup Build(\underline{m}, R)(R)$
  where $Build(\underline{m}, R)(R) = \cup_{m \in \underline{m}} Build\text{-}match\,(m, R)(R)$.
- *For all sets of matches $\underline{m} : L \Rightarrow G$ and $\underline{p} : L \Rightarrow H$:*
  $Union(\underline{m}, \underline{p}) = (\underline{m} : L \Rightarrow G \cup H) \cup (\underline{p} : L \Rightarrow G \cup H) : L \Rightarrow G \cup H$.

Note that we could handle other kinds of graphs in the same way. Here, the *Bind* operation illustrates the interest of accepting isolated nodes in graphs as done in Definition 1 contrary to RDF graphs.

Example 4. As in Example 2 consider the set of matches $\underline{m}_{ex} : L_{ex} \Rightarrow G_{ex}$. We know from Example 3 that the value of count($?s$) with respect to $\underline{m}_{ex}$ is $(3, 3, 3)$ and that the value of count($?s$ by $?p$) with respect to $\underline{m}_{ex}$ is $(2, 2, 1)$.
Thus, $\mathit{Bind}\,(\underline{m}_{ex}, \text{count}(?s), ?n) = \underline{m}'_{ex} : L_{ex} \cup \{?n\} \Rightarrow G_{ex} \cup \{3\}$ and $\mathit{Build}\,(\underline{m}'_{ex}, \{?n\}) = \underline{m}''_{ex} : \{?n\} \Rightarrow G_{ex} \cup \{3\}$ with assignment tables:

$\mathit{Tab}(\underline{m}'_{ex}) =$

$?p$	$?s$	$?c$	$?n$
Alice	Charly	Maths	3
Alice	David	Maths	3
Bob	Eva	CS	3

$\mathit{Tab}(\underline{m}''_{ex}) =$

$?n$
3

$\mathit{Bind}\,(\underline{m}_{ex}, \text{count}(?s \text{ by } ?p), ?n) = \underline{n}'_{ex} : L_{ex} \cup \{?n\} \Rightarrow G_{ex} \cup \{2, 1\}$ with $\mathit{Tab}(\underline{n}'_{ex})$ below. Now let $R'_{ex} = \{(?p, \text{supervises}, ?n)\}$, then $\mathit{Build}\,(\underline{n}'_{ex}, R'_{ex}) = \underline{n}''_{ex} : R'_{ex} \Rightarrow G_{ex} \cup \{(\text{Alice}, \text{supervises}, 2), (\text{Bob}, \text{supervises}, 1)\}$ with assignment tables:

$\mathit{Tab}(\underline{n}'_{ex}) =$

$?p$	$?s$	$?c$	$?n$
Alice	Charly	Maths	2
Alice	David	Maths	2
Bob	Eva	CS	1

$\mathit{Tab}(\underline{n}''_{ex}) =$

$?p$	$?n$
Alice	2
Bob	1

3 Patterns and Queries

The syntax of graph databases is still evolving. We do not consider all technical syntactic details of a real-world language nor all possible constraints on matches. We focus on a core language. Its syntax reflects significant aspects of graph queries. Conditions on graph paths, which can be seen as constraints on matches, are omitted in this paper in order not to make the syntax too cumbersome. We consider mainly two syntactic categories: *patterns* and *queries*, in addition to *expressions* already mentioned. Queries are either SELECT queries, as in most query languages or CONSTRUCT queries, as in SPARQL and G-CORE. A SELECT query applied to a graph returns a *table* which describes a multiset of *solutions* or variable bindings, while a CONSTRUCT query applied to a graph returns a graph. Besides that, a pattern applied to a graph returns a set of matches. Patterns are the basic blocks for building queries. They are defined in Section 3.1 together with their semantics. Queries are defined in Section 3.2 and their semantics is easily derived from the semantics of patterns. In this Section, as in Section 2, the set of *labels* $\mathcal{L}$ is the union of the disjoint sets $\mathcal{C}$ and $\mathcal{V}$, of *constants* and *variables* respectively. We assume that the set $\mathcal{C}$ of constants contains the numbers and strings and the boolean values *true* and *false*.

3.1 Patterns

In Definition 9, the signature for patterns is built by extending the signature Σ_{gq} with a sort Pat for patterns and several operators involving patterns. For instance the operator BASIC in the term $\text{BASIC}\,(L)$ turns a graph L into a pattern. Other operators such as JOIN, BIND or FILTER are rather classical

and were inspired by existing database query languages. Operator BUILD is specific to graph-to-graph queries. The following definition of patterns can be enriched by more specific operators if needed. The formal semantics of patterns is given by an evaluation function in Definition 10.

Definition 9 (syntax of patterns). *The signature Σ_{gq} is extended with a sort* Pat *for* patterns *and the following operators:*

- *If P is a pattern then $[P]$ is a graph, called the* scope graph *of P.*
- *The symbol $\square$ is a pattern, called the* empty pattern*,*
- *If L is a graph then $P = \text{BASIC}\,(L)$ is a pattern, called a* basic pattern*.*
- *If P_1 and P_2 are patterns then $P = P_1 \,\text{JOIN}\, P_2$ is a pattern.*
- *If P_1 is a pattern, e an expression such that $\mathcal{V}(e) \subseteq \mathcal{V}([P_1])$ and x a variable such that $x \notin \mathcal{V}([P_1])$ then $P = P_1 \,\text{BIND}\, e \,\text{AS}\, x$ is a pattern.*
- *If P_1 is a pattern and e an expression such that $\mathcal{V}(e) \subseteq \mathcal{V}([P_1])$ then $P = P_1 \,\text{FILTER}\, e$ is a pattern.*
- *If P_1 is a pattern and R a graph then $P = P_1 \,\text{BUILD}\; R$ is a pattern.*
- *If P_1 and P_2 are patterns such that $[P_1] = [P_2]$ then $P = P_1 \,\text{UNION}\, P_2$ is a pattern.*

The semantics of a pattern P over a graph G is a set of matches $[[P]]_G : [P] \Rightarrow G^{(P)}$ where the source of matches is $[P]$, the so-called scope graph of P and the target graph is $G^{(P)}$. The target graph is obtained by transforming the initial graph G according to the shape of pattern P. These notions are made precise in the following Definition 10. The different pattern operations defined in Definition 9 could be seen as elementary actions that may be used to transform graph databases while computing sets of matches. The induced graph transformation by patterns is similar to traditional algebraic graph transformation processes like DPO [14], AGREE [13] etc. in the following sense. An algebraic graph transformation process does not only transform a graph G into a graph H, but it also transforms an instance of a left-hand side graph L (of a rule) in G into an instance of a right-hand side graph R in H. Such instances are similar to matches. Moreover, patterns are interpreted as sets of matches, so that the induced graph transformation can be seen as a kind of conflict-free simultaneous "parallel" graph transformation process [16].

Definition 10 (evaluation of patterns, set of solutions). *The* set of solutions *or the* value *of a pattern P over a graph G is a set of matches $[[P]]_G : [P] \Rightarrow G^{(P)}$ from the scope graph $[P]$ of P to a graph $G^{(P)}$ that contains G. This value $[[P]]_G : [P] \Rightarrow G^{(P)}$ is defined inductively as follows:*

- $[[\square]]_G = \varnothing_G : \emptyset \Rightarrow G$.
- $[[\text{BASIC}\,(L)]]_G = \mathit{Match}\,(L, G) : L \Rightarrow G$
- $[[P_1 \,\text{JOIN}\, P_2]]_G = \mathit{Join}\,([[P_1]]_G, [[P_2]]_{G^{(P_1)}}) : [P_1] \cup [P_2] \Rightarrow {G^{(P_1)}}^{(P_2)}$
- $[[P_1 \,\text{BIND}\, e \,\text{AS}\, x]]_G = \mathit{Bind}\,([[P_1]]_G, e, x) : [P_1] \cup \{x\} \Rightarrow G^{(P_1)} \cup \{m(e)_{\underline{m}} \mid m \in \underline{m}\}$
  where $\underline{m} = [[P_1]]_G$
- $[[P_1 \,\text{FILTER}\, e]]_G = \mathit{Filter}\,([[P_1]]_G, e) : [P_1] \Rightarrow G^{(P_1)}$

– $[[P_1 \text{ BUILD } R]]_G = \mathit{Build}\,([[P_1]]_G, R) : R \Rightarrow G^{(P_1)} \cup [[P_1]]_G(R)$
– $[[P_1 \text{ UNION } P_2]]_G = \mathit{Union}\,([[P_1]]_G, [[P_2]]_{G^{(P_1)}}) : [P_1] \Rightarrow G^{(P_1)(P_2)}$.

Remark 1. In all cases, the graph $G^{(P)}$ is built by adding to G "whatever is required" for the evaluation. When P is the empty pattern, the value of P over G is the empty subset $\varnothing_G$ of $\mathit{Match}\,(\emptyset, G)$. When P is a BIND, isolated nodes have to be added to G, justifying the use of isolated nodes in graphs.

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 BUILD). Semantically, for every pattern $P = P_1 \text{ OP } \mathit{param}$, let us 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 necessary to evaluate $\underline{m}_1$ before evaluating *param*: 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 BUILD because of the definition of *Build*. 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. Given a pattern $P = P_1 \text{ OP } \mathit{param}$, the pattern P_1 is called 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).

Definition 11. *For every pattern P, the set $\mathcal{V}(P)$ of* in-scope variables *of P is the set $\mathcal{V}([P])$ of variables of the scope graph $[P]$. An expression e is* over *a pattern P if $\mathcal{V}(e) \subseteq \mathcal{V}(P)$.*

Example 5. Let R_{ex} be the graph $R_{ex} = \{(?p, \text{teaches}, ?z), (?s, \text{studies}, ?z)\}$, where $?p$, $?z$ and $?s$ are variables. Note that R_{ex} is the same as L_{ex} (in Example 2), except for the name of one variable. In order to determine when professor $?p$ teaches some topic which is studied by student $?s$, whatever the topic, we consider the following pattern P_{ex}.

$$\begin{aligned} P_{ex} &= \text{BASIC}\,(L_{ex})\,\text{BUILD}\,R_{ex} \\ &= \text{BASIC}\,(\,\{(?p, \text{teaches}, ?t), (?s, \text{studies}, ?t)\}\,) \\ &\quad\ \text{BUILD}\,\{(?p, \text{teaches}, ?z), (?s, \text{studies}, ?z)\} \end{aligned}$$

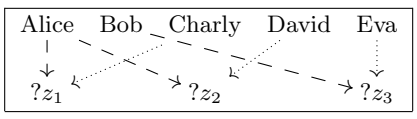
Note that the variable $?z$ in R_{ex} does not appear in L_{ex}. Since there are three matches from L_{ex} to G_{ex} (Example 2), the value of P_{ex} over G_{ex} is:

$$\underline{p}_{ex} : R_{ex} \Rightarrow G'_{ex} \quad \text{with} \quad \mathit{Tab}(\underline{p}_{ex}) =$$

$?p$	$?z$	$?s$
Alice	$?z_1$	Charly
Alice	$?z_2$	David
Bob	$?z_3$	Eva

where $?z_1$, $?z_2$ and $?z_3$ are three fresh variables and G'_{ex} is the union of G_{ex} with the following graph H_{ex}:

(Alice, teaches, $?z_1$) (Charly, studies, $?z_1$)
(Alice, teaches, $?z_2$) (David, studies, $?z_2$)
(Bob, teaches, $?z_3$) (Eva, studies, $?z_3$)



3.2 Queries

We consider two kinds of queries: CONSTRUCT queries, specific to graph database languages, as one may find in SPARQL or G-CORE and SELECT queries, rather classical, close to SQL language, also called MATCH queries in some languages such as Cypher. The semantics of queries is defined from the semantics of patterns. According to Definition 10, all patterns have a graph-to-set-of-matches semantics. In contrast, CONSTRUCT queries have a graph-to-graph semantics and SELECT queries have a graph-to-multiset-of-solutions or graph-to-table semantics.

Definition 12 (syntax of queries). *Let S be a set of variables, R a graph and P a pattern. A* query *Q has one of the following shapes:*
either CONSTRUCT R WHERE P *or* SELECT S WHERE P

Definition 13 (result of CONSTRUCT queries). *Given a pattern P_1 and a graph R, consider the query $Q =$* CONSTRUCT R WHERE P_1 *and the pattern $P = P_1$* BUILD R. *The* result *of the query Q over a graph G, denoted $Result_C(Q, G)$, is the subgraph of $G^{(P)}$ image of R by the set of matches $[[P]]_G$.*

Thus, the result of a CONSTRUCT query Q over a graph G is the graph $Result_C(Q, G) = [[P]]_G(R)$ built by "gluing" the graphs $m(R)$ for all matches m in $[[P]]_G$, where $m(R)$ is a copy of R with each variable $x \in \mathcal{V}(R) - \mathcal{V}(P)$ replaced by a fresh variable (which means, fresh for each m and each x).

Example 6. Consider the query:

$$\begin{aligned} Q_{C,ex} = &\text{ CONSTRUCT } R_{ex} \text{ WHERE BASIC } (L_{ex}) \\ = &\text{ CONSTRUCT } \{ (?p, \text{teaches}, ?z), (?s, \text{studies}, ?z) \} \\ &\text{ WHERE BASIC } (\{ (?p, \text{teaches}, ?t), (?s, \text{studies}, ?t) \}) \end{aligned}$$

The corresponding pattern P_{ex} and the value $\underline{p}_{ex} : R_{ex} \Rightarrow G'_{ex}$ of P_{ex} over G_{ex} are as in Example 5. It follows that the result of the query $Q_{C,ex}$ over G_{ex} is the subgraph of G'_{ex} image of R_{ex} by $\underline{p}_{ex}$: it is the graph H_{ex} from Example 5.

Remark 2. CONSTRUCT queries in SPARQL are similar to CONSTRUCT queries considered in this paper: the variables in $\mathcal{V}(R) - \mathcal{V}(P_1)$ play the same role as the blank nodes in SPARQL. By considering BUILD patterns, thanks to the functional orientation of the definition of patterns, our language allows BUILD subpatterns: this is new and specific to the present study.

For SELECT queries we proceed as for CONSTRUCT queries: we define a transformation from each SELECT query Q to a BUILD pattern P and a transformation from the result of pattern P to the result of query Q. Definition 14 below would deserve more explanations. However this is not the subject of this paper, see [18] for details about how turning a table into a graph (reification).

Definition 14 (result of SELECT queries). *For every set of variables $S = \{s_1, ..., s_n\}$, let $Gr(S)$ denote the graph made of the triples (r, c_j, s_j) for $j \in$*

$\{1, ..., n\}$ *where* r *is a fresh variable and* c_j *is a fresh constant string for each* j*. Given a pattern* P_1 *and a set of variables* $S = \{s_1, ..., s_n\}$ *consider the query* $Q =$ SELECT S WHERE P_1 *and the pattern* $P = P_1$ BUILD $Gr(S)$*. The value of* P *over a graph* G *is a set of matches* $[[P]]_G$ *whose assignment table has* $n+1$ *columns, corresponding to the variables* $r, s_1, ..., s_n$*. The* result *of the query* Q *over a graph* G*, denoted* $Result_S(Q, G)$*, is the multiset of solutions made of the rows of the assignment table of* $[[P]]_G$ *after dropping the column* $?r$*.*

Example 7. Consider the query:

$$Q_{S,ex} = \text{ SELECT } \{?p, ?s\} \text{ WHERE } \text{ BASIC } (L_{ex})$$

Let $R_{S,ex} = Gr(\{?p, ?s\}) = \{(?r, A_p, ?p), (?r, A_s, ?s)\}$ where $?r$ is a fresh variable and A_p, A_s are fresh distinct strings. Then the corresponding pattern is:

$$P_{S,ex} = \text{ BASIC } (L_{ex}) \text{ BUILD } R_{S,ex}$$

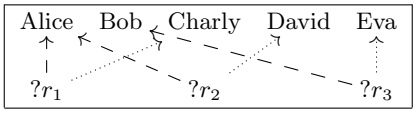
The value of $P_{S,ex}$ over G_{ex} is:

$$\underline{p}_{S,ex} : R_{S,ex} \Rightarrow G'_{S,ex} \text{ with } Tab(\underline{p}_{S,ex}) =$$

$?r$	$?p$	$?s$
$?r_1$	Alice	Charly
$?r_2$	Alice	David
$?r_3$	Bob	Eva

where $?r_1$, $?r_2$ and $?r_3$ are three fresh variables and $G'_{S,ex}$ is the union of G_{ex} with the following graph $H_{S,ex}$:

($?r_1$, A_p, Alice) ($?r_1$, A_s, Charly)
($?r_2$, A_p, Alice) ($?r_2$, A_s, David)
($?r_3$, A_p, Bob) ($?r_3$, A_s, Eva)



It follows that:

$$Result_S(Q_{S,ex}, G_{ex}) =$$

$?p$	$?s$
Alice	Charly
Alice	David
Bob	Eva

4 A Sound and Complete Calculus

In this section we propose a calculus for *solving* patterns and queries based on a relation over patterns called *gq-narrowing*. It computes sets of solutions of patterns (Definition 10) and results of queries (Definitions 13 and 14) over any graph. This calculus is sound and complete with respect to the set-theoretic semantics given in Section 3. It is based on the notion of *configuration* (Definition 15) and a function *Solve* for transforming configurations, which is defined by a rewriting system $\mathcal{R}_{gq}$ (Fig. 1).

In logic-oriented programming languages, narrowing [4] or resolution [30] derivations are used to solve goals and may have the following shape :

$$g_0 \rightsquigarrow_{[\sigma_0]} g_1 \rightsquigarrow_{[\sigma_1]} g_2 \cdots g_n \rightsquigarrow_{[\sigma_n]} g_{n+1}$$

where g_0 is the initial goal to solve (e.g., conjunction of atoms, equations or a (boolean) term), g_{n+1} is a "terminal" goal such as the empty clause, unifiable equations or the constant *true* and g_{i+1} is deduced from g_i by using a clause or a rule in the considered program via subtitution σ_i. From such a derivation, a solution is obtained by simple composition of local substitutions $\sigma_n \circ \ldots \sigma_1 \circ \sigma_0$ with restriction to variables of the initial goal g_0.

In this paper, g_0 is a configuration and the underlying program is a set of rewriting rules augmented by the graph of a considered database. This rewriting system $\mathcal{R}_{gq}$ defines the behavior of the function *Solve* for rewriting configurations. Then three functions are easily derived from *Solve*: $Solve_P$ for solving patterns, $Solve_C$ and $Solve_S$ for solving CONSTRUCT and SELECT queries, respectively.

An important difference between the setting developed in this paper and classical logic-oriented languages comes from the use of functional composition "$\circ$" in $\sigma_n \circ \ldots \sigma_1 \circ \sigma_0$. Depending on the shape of the considered patterns, solutions can be obtained by using additional composition operators such as Join (Definitions 8 and 10) which composes only compatible substitutions computed by different parts of a derivation (e.g., $Join(\sigma_k \circ \ldots \sigma_0, \sigma_n \circ \ldots \sigma_{k+1})$).

Remember from the previous sections that $\emptyset$ is the empty graph, $\underline{i}_G = Match(\emptyset, G) = \{\emptyset_G : \emptyset \to G\}$ and $\square$ is the empty pattern.

Definition 15 (configuration). *A* configuration $[\, P, \underline{m} \,]$ *is made of a pattern* P *and a set of matches* $\underline{m} : L \Rightarrow G$. *Let Config denote the set of configurations. An* initial *configuration is of the form* $[\, P, \underline{i}_G : \emptyset \Rightarrow G \,]$ *for some graph* G *and a* terminal *configuration is of the form* $[\, \square, \underline{m} \,]$.

Roughly speaking, a configuration $[\, P, \underline{m} : L \Rightarrow G \,]$ represents a state where the considered pattern is P and the current graph database is G, which is the target of the current set of matches $\underline{m}$. In this section we define a function:

$Solve : Config \to Config$ by a rewriting system $\mathcal{R}_{gq}$.

This function gives rise to three functions $Solve_P$, $Solve_C$ and $Solve_S$ such that:

$Solve_P(P, G) = [[P]]_G$ for every pattern P and graph G,

$Solve_C(Q, G) = Result_C(Q, G)$ for every CONSTRUCT query Q and graph G,

and $Solve_S(Q, G) = Result_S(Q, G)$ for every SELECT query Q and graph G.

For patterns this runs as follows: in order to find the set of solutions $[[P]]_G$ of a pattern P over a graph G we start from the initial configuration $[\, P, \underline{i}_G \,]$ and we apply the rewriting system $\mathcal{R}_{gq}$ to $Solve([\, P, \underline{i}_G \,])$ until we reach a terminal configuration $[\, \square, \underline{m} : L \Rightarrow G' \,]$, then $\underline{m}$ is the value $[[P]]_G$ of P over G. Notice that graph G' contains G but it is not necessarily equal to G.

In Fig. 1, we provide the rewriting system $\mathcal{R}_{gq}$ which defines the function *Solve*. This function is defined by structural induction on the first component of configurations, i.e., on the patterns. The second argument of configurations,

i.e., the sets of matches, in the left-hand sides of the rules, is always a variable of the form $\underline{m} : L \Rightarrow G$ or simply $\underline{m}$. This variable can be handled easily in the pattern-matching process of the left-hand sides of the rules: there is no need to use higher-order pattern-matching nor unification. In the rules of $\mathcal{R}_{gq}$, the letters P, P_1 and P_2 are variables ranging over *patterns* (sort Pat) while variables L, G and R are ranging over *graphs* (sort Gr) and $\emptyset$ is the constant denoting the empty graph. Symbol e is a variable of sort Exp and x is a variable of subsort Var while $\underline{m}$, $\underline{m}'$ and $\underline{p}$ are variables ranging over sets of matches (sort Som). The rules of Fig. 1 are not dedicated to the graphs used in this paper (Definition 1) but are rather parameterized by the kind of graphs and their corresponding homomorphisms. Indeed, rule r_1 needs the computation of possible matches between two graphs (cf. $Match(L, G)$). The nature of graphs is not specified. They can be RDF graphs, Property graphs, attributed oriented graphs, attributed hypergraphs, constrained graphs etc. The other rules use some operations already introduced in Definition 8, such as *Match*, *Join*, *Bind*, *Filter*, *Build* and *Union*. These operations are to be straightforwardly tuned according to the considered definitions of graphs and graph homomorphisms.

Fig. 1. $\mathcal{R}_{gq}$: Rewriting rules for patterns

$$
\begin{array}{lll}
r_0 & : & Solve([\,\square,\ \underline{m} : L \Rightarrow G\,]) \to [\,\square,\ \varnothing_G : \emptyset \Rightarrow G\,] \\
r_1 & : & Solve([\,\text{BASIC}(L),\ \underline{m} : L \Rightarrow G\,]) \to [\,\square,\ \underline{p}\,] \ \text{ where } \underline{p} = Match(L, G) \\
r_2 & : & Solve([\,P_1 \text{ JOIN } P_2,\ \underline{m}\,]) \to Solve_{JL}(Solve([\,P_1,\ \underline{m}\,]), P_2) \\
r_3 & : & Solve_{JL}([\,\square,\ \underline{m}\,], P) \to Solve_{JR}(\underline{m}, Solve([\,P,\ \underline{m}\,])) \\
r_4 & : & Solve_{JR}(\underline{m}, [\,\square,\ \underline{m}'\,]) \to [\,\square,\ \underline{p}\,] \ \text{ where } \underline{p} = Join(\underline{m}, \underline{m}') \\
r_5 & : & Solve([\,P \text{ BIND } e \text{ AS } x,\ \underline{m}\,]) \to Solve_{BI}(Solve([\,P,\ \underline{m}\,]), e, x) \\
r_6 & : & Solve_{BI}([\,\square,\ \underline{m}\,], e, x) \to [\,\square,\ \underline{p}\,] \ \text{ where } \underline{p} = Bind(\underline{m}, e, x) \\
r_7 & : & Solve([\,P \text{ FILTER } e,\ \underline{m}\,]) \to Solve_{FR}(Solve([\,P,\ \underline{m}\,]), e) \\
r_8 & : & Solve_{FR}([\,\square,\ \underline{m}\,], e) \to [\,\square,\ \underline{p}\,] \ \text{ where } \underline{p} = Filter(\underline{m}, e) \\
r_9 & : & Solve([\,P \text{ BUILD } R,\ \underline{m}\,]) \to Solve_{BU}(Solve([\,P,\ \underline{m}\,]), R) \\
r_{10} & : & Solve_{BU}([\,\square,\ \underline{m}\,], R) \to [\,\square,\ \underline{p}\,] \ \text{ where } \underline{p} = Build(\underline{m}, R) \\
r_{11} & : & Solve([\,P_1 \text{ UNION } P_2,\ \underline{m}\,]) \to Solve_{UL}(Solve([\,P_1,\ \underline{m}\,]), P_2) \\
r_{12} & : & Solve_{UL}([\,\square,\ \underline{m}\,], P) \to Solve_{UR}(\underline{m}, Solve([\,P,\ \underline{m}\,])) \\
r_{13} & : & Solve_{UR}(\underline{m}, [\,\square,\ \underline{m}'\,]) \to [\,\square,\ \underline{p}\,] \ \text{ where } \underline{p} = Union(\underline{m}, \underline{m}')
\end{array}
$$

Rule r_0 considers the degenerated case when one looks for solutions of the empty pattern $\square$. In this case there is no solution and the empty set of matches $\varnothing_G$ is computed. Rule r_1 is key in this calculus because it considers basic patterns of the form BASIC(L) where L is a graph which may contain variables. In this case $Solve([\,\text{BASIC}(L),\ \underline{m}\,])$ consists in finding all matches from L to G. These matches can instantiate variables in L. Thus, the constraint $\underline{p} = Match(L, G)$ of rule r_1 instantiates variables occurring in graph L. This variable instantiation process is close to the narrowing or the resolution-based calculi.

In the context of functional-logic programming languages, several strategies of narrowing-based procedures have been developed to solve goals including even a needed strategy [4]. In this paper, we do not need all the power of narrowing procedures because manipulated data are mostly flat (mainly constants and variables). Thus the unification process used at every step in the narrowing relation is beyond our needs while simple pattern-matching as in classical rewriting is not enough since variables in patterns P cannot be instantiated by simply rewriting the initial term $Solve([P, \underline{i}_G : \emptyset \Rightarrow G])$. Consequently, we propose hereafter a new relation induced by the above rewriting system that we call gq-narrowing. Before the definition of this relation, we recall briefly some notations about first-order terms. Readers not familiar with such notations may consult, e.g., [5].

Definition 16 (position, subterm replacement, substitution, $t{\downarrow_{gq}}$). *A* position *is a sequence of positive integers identifying a subterm in a term. For a term t, the empty sequence, denoted Λ, identifies t itself. When t is of the form $g(t_1, \ldots, t_n)$, the position $i.p$ of t with $1 \leq i \leq n$ and p a position in t_i, identifies the subterm of t_i at position p. The subterm of t at position p is denoted $t|p$ and the result of replacing the subterm of t at position p with term s is written $t[s]_p$. We write $t{\downarrow_{gq}}$ for the term obtained from t where all expressions of $\mathcal{GQ}$-algebra (i.e., operations such as Join, Bind, Filter, Match, etc.) have been evaluated. A substitution σ is a mapping from variables to terms. When $\sigma(x) = u$ with $u \neq x$, we say that x is in the domain of σ. We write $\sigma(t)$ to denote the extension of the application of σ to a term t which is defined inductively as $\sigma(c) = c$ if c is a constant or c is a variable outside the domain of σ. Otherwise $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.*

We write $\mathcal{P}_{gq}(\mathcal{V})$ for the term algebra over the set of variables $\mathcal{V}$ generated by the operations occurring in the rewriting system $\mathcal{R}_{gq}$.

Definition 17 (gq-narrowing $\rightsquigarrow$). *The rewriting system $\mathcal{R}_{gq}$ defines a binary relation $\rightsquigarrow$ over terms in $\mathcal{P}_{gq}(\mathcal{V})$ that we call* gq-narrowing *relation. We write $t \rightsquigarrow_{[u, lhs \to rhs, \sigma]} t'$ or simply $t \rightsquigarrow t'$ and say that t is* gq-narrowable *to t' iff there exists a rule $lhs \to rhs$ in the rewriting system $\mathcal{R}_{gq}$, a position u in t and a substitution σ such that $\sigma(lhs) = t|_u$ and $t' = t[\sigma(rhs){\downarrow_{gq}}]_u$. Then $\rightsquigarrow^*$ denotes the reflexive and transitive closure of the relation $\rightsquigarrow$.*

Notice that in the definition of term $t' = t[\sigma(rhs){\downarrow_{gq}}]_u$ above, the substitution σ is not applied to t as in narrowing ($\sigma(t[rhs]_u \ {\downarrow_{gq}}$) but only to the right-hand side ($\sigma(rhs)$). This is mainly due to the possible use of additional function composition such as *Join* operation. If we consider again rule r_1 in Fig. 1, t' would be of the following shape $t' = t[\Box, (Match(\sigma(L), G) : \sigma(L) \Rightarrow G){\downarrow_{gq}}]_u$. In this case, the evaluation of *Match* operation instantiates possible variables occurring in the pattern BASIC($\sigma(L)$) just like classical narrowing procedures.

The first nice property of the proposed calculus is termination (Proposition 1). In addition, all gq-narrowing derivation steps for solving patterns are needed since at each step only one position is candidate to a gq-narrowing step (Proposition 2). Last but not least, the proposed calculus is sound and complete with respect to the formal semantics given in Section 3 (Theorems 1 and 2).

Proposition 1 (termination). *The relation $\rightsquigarrow$ is terminating.*

Proposition 2 (determinism). *Let $t_0 \rightsquigarrow t_1 \rightsquigarrow \ldots \rightsquigarrow t_n$ be a gq-narrowing derivation with $t_0 = Solve([\, P, \underline{i}_G \,])$. For all $i \in [0..n-1]$, there exists at most one position u_i in t_i such that t_i is gq-narrowable.*

Theorem 1 (soundness). *Let G be a graph, P a pattern and $\underline{m}$ a set of matches such that $Solve([P, \underline{i}_G]) \rightsquigarrow^* [\Box, \underline{m}]$. Then for all morphisms m in $\underline{m}$, there exists a morphism m' equals to m up to renaming of variables such that m' is in $[[P]]_G$.*

Theorem 2 (completeness). *Let G_1, G_2 and X be graphs, P a pattern and $h : X \Rightarrow G_2$ a match in $[[P]]_{G_1}$. Then there exist graphs G'_2 and X', a set of matches $\underline{m} : X' \Rightarrow G'_2$, a derivation $Solve([P, \underline{i}_{G_1}] \rightsquigarrow^* [\Box, \underline{m}]$ and a match $m : X' \Rightarrow G'_2$ in $\underline{m}$ such that m and h are equal up to variable renaming.*

Now we use the *Solve* function for tackling patterns and queries as in Section 3.2. The following three corollaries are obvious consequences of the above results. Remember that the *value* of a pattern P over a graph G is a set of matches $[[P]]_G$, while the *result* of a query Q over a graph G is a graph $Result_C(Q, G)$ when Q is a CONSTRUCT query and a table $Result_S(Q, G)$ when Q is a SELECT query. For SELECT queries we use the graph $Gr(S)$ associated to the set of variables S as in Definition 14 (see [18] for details).

Corollary 1 (solving patterns). *Let P be a pattern. For every graph G there is exactly one gq-narrowing derivation of the form:*
$Solve([P, \underline{i}_G : \emptyset \Rightarrow G]) \rightsquigarrow^* [\Box, \underline{m}]$
and then $\underline{m}$ is the value $Solve_P(P, G) = [[P]]_G$ of P over G.

Example 8. As in Example 5 we consider the pattern:

$$\begin{aligned} P_{ex} = &\text{ BASIC } (L_{ex}) \text{ BUILD } R_{ex} \\ = &\text{ BASIC } (\{\ (?p, \text{teaches}, ?t), (?s, \text{studies}, ?t)\ \}) \\ &\text{ BUILD } \{\ (?p, \text{teaches}, ?z), (?s, \text{studies}, ?z)\ \} \end{aligned}$$

The gq-narrowing derivation is as follows:

$$\begin{aligned} Solve\,(\,[\,P_{ex}, \underline{i}_{G_{ex}}\,]\,) &\rightsquigarrow_{r_9} Solve_{BU}(Solve\,(\,[\ \text{BASIC } (L_{ex}), \underline{i}_{G_{ex}}\,], R_{ex}\,)\,) \\ &\rightsquigarrow_{r_1} Solve_{BU}(\,[\,\Box,\ Match(L_{ex}, G_{ex})\,], R_{ex}\,) \\ &\rightsquigarrow_{r_{10}} [\,\Box,\ Build(Match(L_{ex}, G_{ex}), R_{ex})\,] \end{aligned}$$

We know from Example 5 that $Build(Match(L_{ex}, G_{ex}), R_{ex}) = \underline{p}_{ex} : R_{ex} \Rightarrow G'_{ex}$, thus we get the required result.

Corollary 2 (solving CONSTRUCT queries). *Let R be a graph and P a pattern. Consider the query $Q =$ CONSTRUCT R WHERE P. For every graph G there is exactly one gq-narrowing derivation of the form:*
$Solve(\,[\,P \text{ BUILD } R, \underline{i}_G\,]\,) \rightsquigarrow^* [\,\Box, \underline{m}\,]$
and then the graph image of R by $\underline{m}$ is the result $Solve_C(Q, G) = Result_C(Q, G)$ of Q over G.

Corollary 3 (solving SELECT queries). *Let S be a set of variables and P a pattern. Consider the query $Q =$* SELECT S WHERE P. *For every graph G there is exactly one gq-narrowing derivation of the form:*

$Solve([\, P \text{ BUILD } Gr(S),\ \underline{i}_G \,]) \rightsquigarrow^* [\,\square,\ \underline{m}\,]$

and then the table obtained by dropping the first column from $Tab(\underline{m})$, as in Definition 14, is the result $Solve_S(Q,G) = Result_S(Q,G)$ of Q over G.

5 Conclusion and Related Work

We propose a sound and compete rule-based calculus for a core graph query language (cf. Fig. 1 and Corollaries 2 and 3). The calculus is generic. We illustrate it on RDF graphs to keep examples concise but it can easily be extended and adapted to various data graphs, e.g. Property Graphs, provided that the notion of matches is well defined (cf. rule r_1 in Fig. 1). The syntax of queries was inspired by current implemented languages such as SPARQL, Cypher or preliminary papers about GQL. We were particularly keen to tackle graph-to-graph queries in addition to classical graph-to-relation queries.

Due to the different outcomes of SELECT and CONSTRUCT queries, the composition of graph queries is not straightforward, which contrasts with the situation in the context of relational databases [28, 15]. A particular query nesting, namely EXISTS subqueries, has been implemented in some languages such as Cypher [23] or PGQL [33]. In this paper, we propose the notion of patterns as the main syntactic means to formulate queries. Patterns are defined as terms (trees) on purpose, in order to make it easier to nest patterns at will. Classical composition of (graph) homomorphisms ensures for free the composition of the semantics of nested patterns. It is only at the end of the resolution of a pattern that one chooses to act as a SELECT query and return a table or to act as a CONSTRUCT query and return a graph. One may also choose to act as both kinds of queries in a novel query form we call CONSELECT in [20] by returning at the same time a table and a graph when a pattern is solved.

The results of this paper can be extended to actual graph query languages. For instance, path variables may be added to the syntax and matches between two graphs L and G, as in rule r_1 of Fig. 1, can be constrained by positive, negative or path constraints and written $Match(L, G, \Phi)$ where Φ represents constraints in a given logic (e.g., [31]).

To our knowledge, the proposed calculus is the first sound and complete rule-based calculus dedicated to graph query languages featuring graph-to-graph queries and aggregation operators. The proposed procedure is terminating and does not develop unnecessary derivations. The reader familiar with rewriting systems would notice that a naive and straightforward operationalization of the formal semantics would lead to a rewriting system with fewer rules than those proposed in Fig. 1 but which is not confluent and not all of its derivations yield sound answers.

Among related work, we quote first the use of declarative (functional and logic) languages in the context of relational databases (see, e.g. [9, 26, 1]). In these

works, the considered databases follow the relational paradigm which differs from the graph-oriented one that we are tackling in this paper. Our aim here is not to make connections between graph query languages and functional logic ones. We are rather interested in investigating formally graph query languages, and particularly in using dedicated rewriting techniques for such languages.

The notion of *pattern* present in this paper is close to the syntactic notions of *clauses* in [23] or *graph patterns* in [3]. For such syntactic notions, some authors associate as semantics sets of variables bindings (tables) as in [23, 32, 22] or simply graphs as in [2]. In our case, we associate both variable bindings and graphs since we associate sets of graph homomorphisms to patterns. This semantics is borrowed from a previous work on formal semantics of graph queries based on category theory [17]. Our semantics allows composition of patterns in a natural way. Such composition of patterns is not easy to catch if the semantics is based only on variable bindings but can be recovered when queries have graph outcomes as in G-CORE [2].

Last but not least, the patterns and queries considered in this paper may be seen as formulas of a logic having graphs as interpretations or models. In [24, 21], a graph logic, called *Nested Graph Conditions* (NGC) has been introduced and used to express conditions on graphs. NGC Formulas can be nested and have graph homomorphisms as semantics just like the patterns considered in the present paper. NGC allows one to state conditions on the shape of graphs. In [36], an extension of NGC to attributed graphs has been proposed. NGC formulas can be used as graph queries but do not provide some of desirable query features such as aggregation operators nor do they allow graph transformations as in CONSTRUCT queries or patterns with BUILD operator. Actually, a pattern with a BUILD operator acts as a formula of a dynamic logic (see, e.g. [6]). An operationalization of NGC graph queries has been proposed in [7] where the authors define a set of rewrite rules based on the PO (pushout) approach to compute query solutions. With [7], we share the same abstract definition of queries [7, Definition 2] in the sense that any query is characterized by a so-called *request graph* which corresponds to the notion of *scope graph* of patterns. However, the notion of a query answer according to [7, Definition 3] is defined as a graph homomorphism having the queried graph G as co-domain. This is a particular case of our definition of a query answer. Indeed, the co-domain of a query answer, as we define in the present paper, is the graph G' equals to the queried graph G augmented by the different actions underlying the BUILD and BIND operators involved in the considered pattern or query. So, the proposed rewriting system in [7] departs from ours. Their rules are well adapted to NGC conditions and thus fail to add new items (nodes or edges) to queried graphs and do not take into account aggregations.

Future work includes an implementation of the proposed calculus as well as the investigation of validation techniques for graph database languages, including verification methods e.g., [10, 11] or test techniques as proposed for example in [29].

References

1. J. M. Almendros-Jiménez and A. Becerra-Terón. A safe relational calculus for functional logic deductive databases. *Electron. Notes Theor. Comput. Sci.*, 86(3):168–204, 2003.
2. R. Angles, M. Arenas, P. Barceló, P. A. Boncz, G. H. L. Fletcher, C. Gutiérrez, T. Lindaaker, M. Paradies, S. Plantikow, J. F. Sequeda, O. van Rest, and H. Voigt. G-CORE: A core for future graph query languages. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432. ACM, 2018.
3. R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.
5. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
6. P. Balbiani, R. Echahed, and A. Herzig. A dynamic logic for termgraph rewriting. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr, editors, *Graph Transformations - 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - - October 2, 2010. Proceedings*, volume 6372 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2010.
7. T. Beyhl, D. Blouin, H. Giese, and L. Lambers. On the operationalization of graph queries with generalized discrimination networks. In R. Echahed and M. Minas, editors, *Graph Transformation - 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5-6, 2016, Proceedings*, volume 9761 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2016.
8. D. Bork, D. Karagiannis, and B. Pittl. A survey of modeling language specification techniques. *Inf. Syst.*, 87, 2020.
9. B. Brassel, M. Hanus, and M. Müller. High-level database programming in curry. In *Proc. of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.
10. J. H. Brenas, R. Echahed, and M. Strecker. Verifying graph transformation systems with description logics. In L. Lambers and J. H. Weber, editors, *Graph Transformation - 11th International Conference, ICGT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-26, 2018, Proceedings*, volume 10887 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2018.
11. J. H. Brenas, R. Echahed, and M. Strecker. Reasoning formally about database queries and updates. In M. H. ter Beek, A. McIver, and J. N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 556–572. Springer, 2019.
12. D. D. Chamberlin and R. F. Boyce. SEQUEL: a structured English query language. In R. Rustin, editor, *FIDET 74': Data models: data-structure-set versus relational: Workshop on Data Description, Access, and Control, May 1–3, 1974, Ann Arbor, Michigan*, pages 249–264, 1974.
13. A. Corradini, D. Duval, R. Echahed, F. Prost, and L. Ribeiro. AGREE - algebraic graph rewriting with controlled embedding. In *Graph Transformation - 8th*

International Conference, ICGT 2015, volume 9151 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2015.
14. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - part I: basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 163–246, 1997.
15. C. J. Date. *A guide to the SQL standard: a user's guide to the standard relational language SQL*. 1987.
16. T. B. de la Tour and R. Echahed. Parallel rewriting of attributed graphs. *Theor. Comput. Sci.*, 848:106–132, 2020.
17. D. Duval, R. Echahed, and F. Prost. An algebraic graph transformation approach for RDF and SPARQL. In B. Hoffmann and M. Minas, editors, *Proceedings of the Eleventh International Workshop on Graph Computation Models, Online-Workshop, 24th June 2020*, volume 330 of *EPTCS*, pages 55–70, 2020.
18. D. Duval, R. Echahed, and F. Prost. All you need is CONSTRUCT. *CoRR*, abs/2010.00843, 2020.
19. D. Duval, R. Echahed, and F. Prost. Querying RDF databases with sub-constructs. In T. Kutsia, editor, *Proceedings of the 9th International Symposium on Symbolic Computation in Software Science, SCSS 2021, Hagenberg, Austria, September 8-10, 2021*, volume 342 of *EPTCS*, pages 49–64, 2021.
20. D. Duval, R. Echahed, and F. Prost. A rule-based operational semantics of graph query languages. *CoRR*, abs/2202.10142, 2022.
21. H. Ehrig, K. Ehrig, A. Habel, and K. Pennemann. Constraints and application conditions: From graphs to high-level structures. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2004.
22. 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.
23. N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD Conference*, pages 1433–1445. ACM, 2018.
24. A. Habel and K. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Math. Struct. Comput. Sci.*, 19(2):245–296, 2009.
25. A. Habel and D. Plump. Complete strategies for term graph narrowing. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT '98, Lisbon, Portugal, April 2-4, 1998, Selected Papers*, volume 1589 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 1998.
26. M. Hanus. Dynamic predicates in functional logic programs. volume 2004. EAPLS, 2004.
27. S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1213–1216. ACM, 2011.
28. W. Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.
29. L. Lambers, S. Schneider, and M. Weisgut. Model-based testing of read only graph queries. In *13th IEEE International Conference on Software Testing, Verification*

and Validation Workshops, ICSTW 2020, Porto, Portugal, October 24-28, 2020, pages 24–34. IEEE, 2020.
30. J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
31. M. Navarro, F. Orejas, E. Pino, and L. Lambers. A navigational logic for reasoning about graph properties. *J. Log. Algebraic Methods Program.*, 118, 2021.
32. J. Pérez, M. Arenas, and C. Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.
33. PGQL 1.5 Specification. https://pgql-lang.org/spec/1.5/, August 2022.
34. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
35. S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. G. Aref, M. Arenas, M. Besta, P. A. Boncz, K. Daudjee, E. D. Valle, S. Dumbrava, O. Hartig, B. Haslhofer, T. Hegeman, J. Hidders, K. Hose, A. Iamnitchi, V. Kalavri, H. Kapp, W. Martens, M. T. Özsu, E. Peukert, S. Plantikow, M. Ragab, M. Ripeanu, S. Salihoglu, C. Schulz, P. Selmer, J. F. Sequeda, J. Shinavier, G. Szárnyas, R. Tommasini, A. Tumeo, A. Uta, A. L. Varbanescu, H. Wu, N. Yakovets, D. Yan, and E. Yoneki. The future is big graphs: a community view on graph processing systems. *Commun. ACM*, 64(9):62–71, 2021.
36. S. Schneider, L. Lambers, and F. Orejas. Automated reasoning for attributed graph properties. *Int. J. Softw. Tools Technol. Transf.*, 20(6):705–737, 2018.
37. SPARQL 1.1 Query Language. W3C Recommendation, March 2013.
38. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, February 2014.