Symbolic Graph Query Solving

Dominique Duval, Rachid Echahed

Symbolic Graph Query Solving *

Dominique Duval and Rachid Echahed

CNRS and University Grenoble Alpes, Grenoble, France
dominique.duval@imag.fr and rachid.echahed@imag.fr

Abstract. Property graphs play an important role in representing data bases in modern graph query languages. In this paper we focus on a particular class of graph queries known as Graph-to-Graph or CONSTRUCT queries. Such queries return graphs instead of tables when applied to actual Property Graphs. We propose a symbolic graph query solving procedure which combines rewriting techniques for goal solving as well as graph transformation techniques. The proposed procedure is proven sound and complete.

Keywords: Operational semantics, Rewrite systems, Graph query languages

1 Introduction

Graph database systems are becoming more and more in demand thanks to their legibility and high flexibility. Several graph query languages have been proposed recently or being developed such as Cypher [8], PGQL[1], GSQL[2] and G-CORE [1], inducing an ISO project to standardize a graph query language. Two separate standards are being issued[3] : GQL and SQL/PGQ.

The aforementioned languages are based on a particular definition of graphs called *property graphs*. These graphs feature nodes and edges that can be typed by means of labels and can also be endowed with finite records. Such graphs provide great flexibility in data representation.

Property graphs are also used to formulate queries. We focus in this paper on a special kind of graph queries, known as graph-to-graph queries or CONSTRUCT queries [11, 1], which yield a graph as a result, when applied to a graph data, instead of classical tables. This is by no means a limitation since it is well know that tables can easily be encoded as graphs (see, e.g. [5]).

The semantics of graph queries are based on variables bindings in general [2, 8]. Such semantics are not appropriate to handle easily query nesting. Recently, we proposed in [7], in the context of RDF-graphs [12], a kernel query language whose semantics is based on graph homomorphisms. Composition of

* Partly supported by the French ANR project VERIGRAPH # ANR-21-CE48-0015
[1] https://pgql-lang.org/
[2] https://www.tigergraph.com/gsql/
[3] https://www.gqlstandards.org/

homomorphisms ensures in an obvious way query nesting or more precisely *pattern* nesting. In the present paper we follow such semantics and consider a core query language which allows nesting of patterns in the context of the rich structures of property graphs. We also propose a new symbolic procedure based on a rewriting system which incorporates operators that have been tailored expressly for property graph structures. The proposed procedure, which is proven to be deterministic, terminating, sound and complete, solves queries in a way close to narrowing-based procedures [3], but limited to specific operators over data graphs represented as property graphs.

The paper first defines, in the next section, the data graphs considered in the paper together with some additional definitions. In Section 3, the notions of patterns and CONSTRUCT queries are introduced by their syntax and semantics. Then, in Section 4, a rewriting system defining a procedure to solve patterns and queries is introduced and the main properties of the procedure are stated. Concluding remarks are provided in Section 5.

2 Data Graphs

In this section, we start by fixing some definitions of the data structures used in this paper. We assume given a finite set Lab of labels, a finite set $Prop$ of properties and a possibly infinite set Val of values (integers, booleans, strings, etc.). In the rest of the paper we assume that all expressions are well typed. Let A be a possibly infinite set, we denote by $\widehat{\mathcal{P}}(A)$ the set of finite subsets of A.

Below, we provide the definition of property graphs we consider in this paper.

Definition 1 (Property Graph).
A property graph G is a tuple $G = (N, E, src, tgt, \lambda, \pi)$ where N is a finite set of nodes, E is a finite set of edges such that $N \cap E = \emptyset$. src (resp. tgt) is a function $src : E \to N$ (resp. $tgt : E \to N$) defining the source (resp. target) of edges. The function $\lambda : N \cup E \to \mathcal{P}(Lab)$ specifies, for every node or edge, possible labels in Lab. The function $\pi : N \cup E \to \widehat{\mathcal{P}}(Prop \times Val)$ specifies, for each node or edge, the possible finite property-value associations.

Example 1. We introduce here a running example of a property graph representing the authorship relation between *persons* and *papers*. The proposed sample graph $G1 = (N_1, E_1, src_1, tgt_1, \lambda_1, \pi_1)$, depicted in Fig. 1, is such that: $N = \{ni \mid 1 \leq i \leq 6\}$, $E_1 = \{ei \mid 1 \leq i \leq 5\}$, $\lambda_1(n1) = \lambda_1(n2) = \lambda_1(n3) = \{\text{person}\}$ and $\lambda_1(n4) = \lambda_1(n5) = \lambda_1(n6) = \{\text{paper}\}$. $\lambda_1(ei) = \{\text{author}\}$ for $1 \leq i \leq 5$. The function π_1 does not assign any data to edges, i.e., $\pi_1(ei) = \emptyset$ for $1 \leq i \leq 5$. However, the definition of π_1 is a bit long to write. It can be deduced easily from Fig. 1. For instance $\pi_1(n1) = \{(name, Paul\,Erdős), (Inst, Univ.Manchester), (Inst, IAS(Princeton))\}$.

The notion of paths plays a significant role in graph queries.

Definition 2 (Paths).
Let $G = (N, E, src, tgt, \lambda, \pi)$ be a property graph. A path p in G is a string

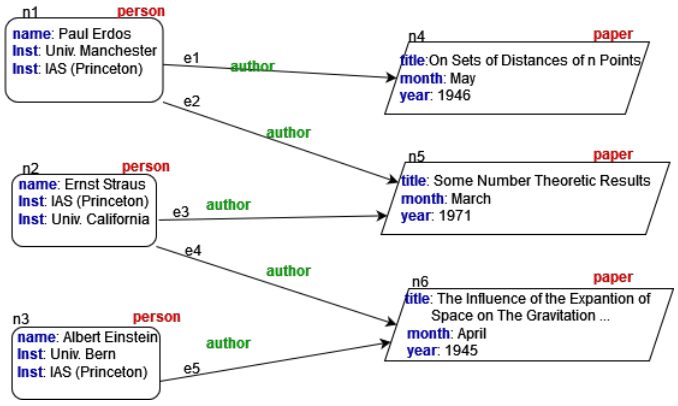

Fig. 1. Sample Property Graph, $G1$, of authors and papers

over $N \cup E$ of the form $n_1e_1n_2 \dots n_ke_kn_{k+1}$ for some $k \geq 0$ such that for all j, $1 \leq j \leq k$, $src(e_j) = n_j$ and $tgt(e_j) = n_{j+1}$. We denote by $\mathcal{P}th(G)$ the set of paths of graph G. The source (resp. target) of a path of the form $n_1e_1n_2 \dots n_ke_kn_{k+1}$ is n_1 which is equal to $src(e_1)$ (resp. n_{k+1}, which is equal to $tgt(e_k)$). The concatenation of two paths p_1 and p_2 is written p_1+p_2 and assumes that $src(p_2) = tgt(p_1)$. That is, $(n_1e_1n_2 \dots n_ke_kn_{k+1}) + (n'_1e'_1n'_2 \dots n'_qe'_qn'_{q+1}) = n_1e_1n_2 \dots n_ke_kn'_1e'_1n'_2 \dots n'_qe'_qn'_{q+1}$ if $n_{k+1} = n'_1$.

Definition 3 (Property Graphs with Paths(PGP)).
A property graph with paths (PGP) G is a tuple $G = (N, E, P, src, tgt, \lambda, \pi)$ where $(N, E, src, tgt, \lambda, \pi)$ is a property graph and P is a finite subset of $\mathcal{P}th(G)$.

Notice that we assume the set of paths P to be finite in the definition above only for decidabilty reasons of the operational semantics presented later.

Let $f_i : A_i \to B_i$ for $i \in \{1, 2\}$ be two functions. We say that f_1 and f_2 are joinable if $f_1(x) = f_2(x)$ for all x in $A_1 \cap A_2$. The two following definitions, which are used later, define the union of graphs and graph quotients.

Definition 4 (Graph Union). *Let $G_i = (N_i, E_i, P_i, src_i, tgt_i, \lambda_i, \pi_i)$ for $i \in \{1, 2\}$ be two PGPs such that the functions src_1 and src_2 (resp. tgt_1 and tgt_2) are joinable and $N_i \cap E_j = \emptyset$ for i, j in $\{1, 2\}$. We denote by $G_1 \cup G_2$ the graph $G_1 \cup G_2 = (N_1 \cup N_2, E_1 \cup E_2, P_1 \cup P_2, src_\cup, tgt_\cup, \lambda_\cup, \pi_\cup)$ such that : (i) for all x in $N_1 \cup N_2 \cup E_1 \cup E_2$, $\lambda_\cup(x) = \lambda_1(x) \cup \lambda_2(x)$ and $\pi_\cup(x) = \pi_1(x) \cup \pi_2(x)$, and (ii) for all x in $E_1 \cup E_2$, $src_\cup(x) = src_1(x)$ (resp. $tgt_\cup(x) = tgt_1(x)$) if x in E_1, otherwise $src_\cup(x) = src_2(x)$ (resp. $tgt_\cup(x) = tgt_2(x)$).*

Definition 5 (Graph equivalence, Quotient graph).
Let $G = (N, E, P, src, tgt, \lambda, \pi)$. A graph equivalence $\cong$ is a pair $(\cong_n, \cong_e)$ of equivalences on nodes $\cong_n$ and on edges $\cong_e$ such that, for all e_1, e_2 in E, $e_1 \cong_e e_2$ implies that $src(e_1) \cong_n src(e_2)$ and $tgt(e_1) \cong_n tgt(e_2)$. The quotient graph $G_\cong =$

$(N_{\cong_n}, E_{\cong_e}, P_{\cong}, \overline{src}, \overline{tgt}, \bar{\lambda}, \bar{\pi})$ is such that $N_{\cong_n}, E_{\cong_e}, P_{\cong}$ are the intended quotient sets and for all $\bar{e}$ in $E_{\cong_e}$, $\overline{src}(\bar{e}) = \overline{(src(e))}$ and $\overline{tgt}(\bar{e}) = \overline{(tgt(e))}$ where $\bar{x}$ denotes the equivalence class of element x. For all $\bar{y} \in (N_{\cong_n} \cup E_{\cong_e})$, $\bar{\lambda}(\bar{y}) = \cup_{i \in \bar{y}} \lambda(i)$ and $\bar{\pi}(\bar{y}) = \cup_{i \in \bar{y}} \pi(i)$.

The notion of homomorphism below is quite direct. One just has to deal carefully with all different notions involved in a property graph.

Definition 6 (PGP Graph homomorphism).
Let $G_i = (N_i, E_i, P_i, src_i, tgt_i, \lambda_i, \pi_i)$ for $i \in \{1, 2\}$ be two PGPs. A graph homomorphism from graph G_1 to graph G_2, denoted $m : G_1 \to G_2$, is defined by two functions $m_n : N_1 \to N_2$ and $m_e : E_1 \to E_2$ such that:

1. *For all a in E_1, $src_2(m_e(a)) = m_n(src_1(a))$ and $tgt_2(m_e(a)) = m_n(tgt_1(a))$.*
2. *For all b in N_1, $\lambda_1(b) \subseteq \lambda_2(m_n(b))$ and For all a in E_1, $\lambda_1(a) \subseteq \lambda_2(m_e(a))$*
3. *For all b in N_1, $\pi_1(b) \subseteq \pi_2(m_n(b))$ and For all a in E_1, $\pi_1(a) \subseteq \pi_2(m_e(a))$*
4. *For all p in P_1, $m_p(p)$ is in P_2 where m_p is defined as follows :*
   $m_p(n_1 e_1 n_2 \dots n_k e_k n_{k+1}) = m_n(n_1) m_e(e_1) m_n(n_2) \dots m_n(n_k) m_e(e_k) m_n(n_{k+1})$

Basic graph queries could be expressed by means of PGPs. For that, PGP graphs have to be endowed with variables which can be instantiated while solving graph queries. In the definition of homomorphisms above, elements in N_1 (resp. E_1) may be matched by any element in N_2 (resp. E_2). However, elements in Lab, $Prop$ and Val behave as constants. Therefore, we assume in the sequel given four sets of variables, elements of which can be used in PGPs. These sets of variables are V_v (resp. V_p, V_l and V_{pth}) which represent sets of variables ranging over the set of values Val (resp. of properties $Prop$, of labels Lab and of paths $\mathcal{P}th(G)$).

We call *path expression* a path which may include variables in V_{pth} and call *open PGP* a PGP including variables.

Definition 7 (Path Expression).
Let G be a property graph. A path expression *over G and V_{pth} is an expression generated by the following grammar where p is a path in $\mathcal{P}th(G)$, x is a path variable in V_{pth} and $+$ stands for the concatenation of paths : $S \to p \mid x \mid S + S$. A path expression, of the form $p_1 + p_2$, resulting of the concatenation of the paths p_1 and p_2 is such that the target of p_1 equals the source of p_2. We write $\mathcal{P}th(G, V_{pth})$ the set of path expressions over graph G.*

Definition 8 (Open PGP).
A PGP graph $G = (N, E, P, src, tgt, \lambda, \pi)$ is said to be an open PGP *if the set P can include path expressions (i.e. $P \subseteq \mathcal{P}th(G, V_{pth})$) and the target sets of functions λ and π can include variables. That is $\lambda : N \cup E \to \mathcal{P}(Lab \cup V_l)$ and $\pi : N \cup E \to \widehat{\mathcal{P}}((Prop \cup V_p) \times (Val \cup V_v))$.*

Now we are ready to define the notion of *match* between open PGPs. A match is a homomorphism based on instantiations of variables. We denote by $\mathcal{V}(G)$ the set of all variables appearing in graph G, $\mathcal{V}_{pth}(G) = \mathcal{V}(G) \cap V_{pth}$ the set of path variables occurring in G, $\mathcal{V}_{lab}(G) = \mathcal{V}(G) \cap V_l$, the set of label variables appearing in G, $\mathcal{V}_{prp}(G) = \mathcal{V}(G) \cap V_p$, the set of property variables appearing in G and $\mathcal{V}_{val}(G) = \mathcal{V}(G) \cap V_v$ the set of value variables appearing in G.

Definition 9 (Match).
Let L and G be two open PGP, $L = (N_L, E_L, P_L, src_L, tgt_L, \lambda_L, \pi_L)$ and $G = (N_G, E_G, P_G, src_G, tgt_G, \lambda_G, \pi_G)$. A match $m : L \to G$ is defined by six functions $m_n : N_L \to N_G$, $m_e : E_L \to E_G$, $m_{lab} : \mathcal{V}_{lab}(L) \to Lab \cup \mathcal{V}_{lab}(G)$, $m_{val} : \mathcal{V}_{val}(L) \to Val \cup \mathcal{V}_{val}(G)$, $m_{prp} : \mathcal{V}_{prp}(L) \to Prop \cup \mathcal{V}_{val}(G)$ and $m_{pth} : \mathcal{V}_{pth}(L) \to \mathcal{P}th(G, \mathcal{V}_{pth}(G))$ such that:

1. *For all a in E_L, $src_G(m_e(a)) = m_n(src_L(a))$ and $tgt_G(m_e(a)) = m_n(tgt_L(a))$.*
2. *For all b in N_L, $m(\lambda_L(b)) \subseteq \lambda_G(m_n(b))$ and $m(\pi_L(b)) \subseteq \pi_G(m_n(b))$.*
3. *For all a in E_L, $m(\lambda_L(a)) \subseteq \lambda_G(m_e(a))$ and $m(\pi_L(a)) \subseteq \pi_G(m_e(a))$.*
4. *For all p in P_L, $m(p)$ is in P_G.*

Where

- *$m(F)$, with $F \subseteq (Lab \cup \mathcal{V}_{lab}(L))$, is defined as $m(F) = \{m(f) \mid f \in F\}$ such that $m(f) = f$ if f is a label in Lab and $m(f) = m_{lab}(f)$ if f is a label variable in $\mathcal{V}_{lab}(L)$.*
- *$m(U)$, with $U \subseteq \widehat{\mathcal{P}}((Prop \cup \mathcal{V}_{prp}(L)) \times (Val \cup \mathcal{V}_{val}(L)))$, is defined as $m(U) = \{(m(u_p), m(u_v)) \mid (u_p, u_v) \in U\}$ with $m(u_p) = u_p$ if u_p is a property in $Prop$ and $m(u_p) = m_{prp}(u_p)$ if u_p is a property variable in $\mathcal{V}_{prp}(L)$ and $m(u_v) = u_v$ if u_v is a value in Val and $m(u_v) = m_{val}(u_v)$ if u_v is a value variable in $\mathcal{V}_{val}(L)$.*
- *$m(p)$, with p being a path expression, is defined as follows: $m(p) = m(u) + m(w)$ if p is the concatenation of two paths $p = u{+}w$; $m(p) = m_{pth}(p)$ if p is a path variable and $m(p) = m_n(n_1)m_e(e_1)m_n(n_2)\dots m_n(n_k)m_e(e_k)m_n(n_{k+1})$ if p is of the form $n_1e_1n_2\dots n_ke_kn_{k+1}$.*

In the sequel, we will drop subscripts and simply write $m(t)$ for the application of a match or a homomorphism m on item t when it is clear from the context.

Definition 10 (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 11 (set of matches). *Let L be an open PGP and G a PGP. A set $\underline{m}$ of matches, all of them from L to G, is denoted $\underline{m} : L \Rightarrow G$. 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 $\emptyset$, the set $Match(\emptyset, G) : \emptyset \Rightarrow G$ has one unique element which is the inclusion of $\emptyset$ into G, then we denote $\underline{i}_G = Match(\emptyset, G) : \emptyset \Rightarrow G$ this one-element set and $\varnothing_G : \emptyset \Rightarrow G$ its empty subset.*

Example 2. Let $L1$ be the following open PGP consisting of two nodes x and y of label *person* and one node z of label *paper*, in addition to two edges $x1$ and $y1$ of label *author*.

```
L1 = (x:person)-[x1:author]->(z:paper)
     (y:person)-[y1:author]->(z:paper)
```

The set of matches $\underline{m} = Match(L1, G1)$ from $L1$ to graph $G1$ of Example 1 consists of 9 matches $\underline{m} = \{m_i \mid 1 \leq i \leq 9\}$ whose variable assignements are depicted in the following table.

	x	y	z	$x1$	$y1$
m_1	n1	n2	n5	e2	e3
m_2	n2	n1	n5	e3	e2
m_3	n2	n3	n6	e4	e5
m_4	n3	n2	n6	e5	e4
m_5	n1	n1	n4	e1	e1
m_6	n1	n1	n5	e2	e2
m_7	n2	n2	n5	e3	e3
m_8	n2	n2	n6	e4	e4
m_9	n3	n3	n6	e5	e5

Definition 12 (Canonical match and Graph equivalence).
If $h : L \to G$ is a match and $\cong$ a graph equivalence on G, we write $h_{\cong} : L \to G_{\cong}$ the canonical match obtained from h and $\cong$. If $\underline{h} : L \to G$ is a set of matches. We write $\underline{h}_{\cong} : L \to G_{\cong}$ the set of matches $h_{\cong} : L \to G_{\cong}$ for every match h in $\underline{h}$.

3 Patterns and Queries

In this section we introduce the class of queries we tackle in this paper, namely the CONSTRUCT queries. As said earlier, such queries easily encode SELECT or MATCH queries since tables can be translated into graphs. We start by introducing some basic notions such as *expressions* and some basic operators on graphs that contribute to define the syntax and the semantics of patterns and queries.

3.1 Expressions

Queries can be endowed with classical expressions on integers, strings, booleans etc. which contribute to filtering solutions or computing parts of them such as aggregation operations.

Let $Op_1 = \{-, \text{NOT}, \ldots\}$, $Op_2 = \{+, -, \times, /, =, >, <, \text{AND}, \text{OR}, \ldots\}$ and $Agg = \{\text{MAX}, \text{MIN}, \text{SUM}, \text{AVG}, \text{COUNT}, \text{SHORTEST}, \text{SIMPLE}, \text{TRAIL}, \ldots\}$. be sets of unary operations (resp. binary operations and aggregation operations).

An *Expression* e and its set of variables $\mathcal{V}(e)$ are defined recursively as follows, where $c \in Val$, $x \in \mathcal{V}_{val}$, y is a node, $p \in (Prop \cup \mathcal{V}_{prp})$, $op_1 \in Op_1$, $op_2 \in Op_2$, $agg \in Agg$:

$e ::= c \mid x \mid y.p \mid op_1\ e \mid e\ op_2\ e \mid agg(e_1)$.

$\mathcal{V}(c) = \varnothing$, $\mathcal{V}(x) = \{x\}$, $\mathcal{V}(op_1\ e) = \mathcal{V}(e)$, $\mathcal{V}(e_1\ op_2\ e_2) = \mathcal{V}(e_1) \cup \mathcal{V}(e_2)$, $\mathcal{V}(agg(e)) = \mathcal{V}(e)$, $\mathcal{V}(y.p) = \{y\} \cup (\{p\} \cap \mathcal{V}_{prp})$.

The *value* of an expression with respect to a set of matches $\underline{m}$ (Definition 13) is a family of values $\underline{ev}(\underline{m}, e) = (ev(\underline{m}, e)_m)_{m \in \underline{m}}$ indexed by the set $\underline{m}$. When the expression e is free from any aggregation operator then $ev(\underline{m}, e)_m$ is simply $m(e)$. But in general $ev(\underline{m}, e)_m$ depends on e and m and it may also depend on other matches in $\underline{m}$ when e involves aggregation operators. To each basic operator *op* is associated a function $[\![op]\!]$ (or simply *op*) from values to values if *op* is unary and from pairs of values to values if *op* is binary. To each aggregation operator *agg* in *Agg* is associated a function $[\![agg]\!]$ (or simply *agg*) from *multisets* of values to values. Note that each family of values determines a multiset of values: for instance a family $\underline{c} = (c_m)_{m \in \underline{m}}$ of values indexed by the elements of a set of matches $\underline{m}$ determines the multiset of values $\{\!|c_m \mid m \in \underline{m}|\!\}$, which is also denoted $\underline{c}$ when there is no ambiguity.

Definition 13 (evaluation of expressions). *Let L be a graph, e an expression over L and $\underline{m} : L \Rightarrow G$ a set of matches. The* value *of e with respect to $\underline{m}$ is the family $\underline{ev}(\underline{m}, e) = (ev(\underline{m}, e)_m)_{m \in \underline{m}}$ defined recursively as follows. It is assumed that each $ev(\underline{m}, e)_m$ in this definition is a value (constant).*

- $ev(\underline{m}, c)_m = c$,
- $ev(\underline{m}, x)_m = m(x)$,
- $ev(\underline{m}, y.p)_m = ev(\underline{m}, e_1)_m$ *if* $(m(p), e_1) \in \pi_G(m(y))$,
- $ev(\underline{m}, op\ e_1)_m = [\![op]\!]\ ev(\underline{m}, e_1)_m$,
- $ev(\underline{m}, e_1\ op\ e_2)_m = ev(\underline{m}, e_1)_m\ [\![op]\!]\ ev(\underline{m}, e_2)_m$,
- $ev(\underline{m}, agg(e_1))_m = [\![agg]\!](\underline{ev}(\underline{m}, e_1))$.

3.2 Operations over Data Graphs and Matches

In graph-to-graph queries, one may have to construct new graphs during the resolution process. Below we define the canonical match $\mathit{Build}(m, R) : R \to G \cup H_{m,R}$ which builds, from a match $m : L \to G$, a new match starting from a given graph R by adding, to graph G, the image of R via m, namely $H_{m,R}$. Intuitively, $H_{m,R}$ is obtained from R by renaming the nodes and edges and the variables which are not instantiated by m. $H_{m,R}$ may share nodes and edges with G according to the definition of m.

Definition 14 (building a match). *Let $m : L \to G$ be a match and R an open PGP with possible expressions as values such that $\mathcal{V}_{pth}(R) \subseteq \mathcal{V}_{pth}(L)$, $\mathcal{V}_{lab}(R) \subseteq \mathcal{V}_{lab}(L)$ and $\mathcal{V}_{prp}(R) \subseteq \mathcal{V}_{prp}(L)$; that is, the path variables, the label variables and the property variables appearing in R also occur in L. The match $\mathit{Build}(m, R) : R \to G \cup H_{m,R}$ is the unique match (up to variable renaming) based on the following six functions $h_n, h_e, h_{lab}, h_{val}, h_{prp}$ and h_{pth} where $R = (N_R, E_R, P_R, src_R, tgt_R, \lambda_R, \pi_R)$ and $H_{m,R} = (N_H, E_H, P_H, src_H, tgt_H, \lambda_H, \pi_H)$ such that:*

- $h_n : N_R \to N_H$ *such that* $h_n(x) = m(x)$ *if* $x \in N_R \cap N_L$ *; otherwise* $h_n(x) = x'$ *where* x' *is a fresh node.*

- $h_e : E_R \to E_H$ *such that* $h_e(x) = m(x)$ *if* $x \in E_R \cap E_L$ *; otherwise* $h_e(x) = x'$ *where* x' *is a fresh edge with* $src_H(x') = h_n(src_R(x))$ *and* $tgt_H(x') = h_n(tgt_R(x))$
- $h_{lab} : \mathcal{V}_{lab}(R) \to Lab \cup \mathcal{V}_{lab}(H_{m,R})$ *such that* $h_{lab}(x) = m(x)$ *if* x *is a variable label in* $\mathcal{V}_{lab}(R)$ *(recall that we assume* $\mathcal{V}_{lab}(R) \subseteq \mathcal{V}_{lab}(L)$*)* .
- $h_{val} : \mathcal{V}_{val}(R) \to Val \cup \mathcal{V}_{val}(H_{m,R})$ *such that* $h_{val}(x) = m(x)$ *if* x *is a value variable in* $\mathcal{V}_{val}(R) \cap \mathcal{V}_{val}(L)$ *; otherwise* $h_{val}(x) = x'$ *where* x' *is a fresh value variable.*
- $h_{prp} : \mathcal{V}_{prp}(R) \to Prop \cup \mathcal{V}_{prp}(H_{m,R})$ *such that* $h_{prp}(x) = m(x)$ *if* x *is a property variable in* $\mathcal{V}_{prp}(R)$ *(recall that we assume* $\mathcal{V}_{prp}(R) \subseteq \mathcal{V}_{prp}(L)$*).*
- $h_{pth} : \mathcal{V}_{pth}(R) \to \mathcal{P}th(H_{m,R}, \mathcal{V}_{pth}(H_{m,R}))$ *such that* $h_{pth}(x) = m(x)$ *(recall that we assume* $\mathcal{V}_{pth}(R) \subseteq \mathcal{V}_{pth}(L)$*).*
- $\lambda_H(h_n(x)) = \{h_{lab}(z) \mid x \in N_R \text{ and } z \in \lambda_R(x)\}$.
- $\lambda_H(h_e(x)) = \{h_{lab}(z) \mid x \in E_R \text{ and } z \in \lambda_R(x)\}$.
- $\pi_H(h_n(x)) = \{(m(p), m(e)) \mid x \in N_R \text{ and } (p, e) \in \pi_R(x)\}$.
- $\pi_H(h_e(x)) = \{(m(p), m(e)) \mid x \in E_R \text{ and } (p, e) \in \pi_R(x)\}$.

Example 3. Let $m_1 : L1 \to G1$ be the match as defined in Example 2 (cf., the first column of the table) where
`L1 = (x:person)-[x1:author]->(z:paper)<-[y1:author]-(y:person)` and $G1$ the graph defined in Example 1. Let `R3 = (x) -[u:coauthor]-> (y)`.

The match $Build(m_1, R3) : R3 \to G1 \cup H_{m_1,R3}$ is rather straightforward. The graph $G1 \cup H_{m_1,R3}$ is obtained from graph $G1$ by simply adding one edge with label `coauthor` from node $n1$ to node $n2$. Notice that when one considers all the 9 matches of Example 2 and apply $Build(m_i, R3)$ for each match m_i with $1 \leq i \leq 9$, one gets graph $G3$ as illustrated in Example 4.

Notation: Let $G = (N, E, P, src, tgt, \lambda, \pi)$ be a PGP and x an element in $N \cup E$, p a property in $Prop$ and v a value in Val. We denote by $G[x.p \leftarrow v]$ the PGP $(N, E, P, src, tgt, \lambda, \pi')$ such that for all i in $N \cup E$, $\pi'(i) = \pi(i)$ if $i \neq x$ and $\pi'(x) = \pi(x) \cup \{(p, v)\}$.

Hereafter, we define the main operations over graphs and sets of matches that we use in the definition of the semantics of patterns and thus for queries.

Definition 15 (Operations over Graphs and Sets of Matches).

- *For all open PGP graph* L *and PGP graph* G*:*
  $Match(L, G) : L \Rightarrow G$ *is the set of all matches from* L *to* G*.*
- *For every set of matches* $\underline{m} : L \Rightarrow G$ *and every expression* e*:*
  $Filter(\underline{m}, e) = \{m \mid m \in \underline{m} \wedge ev(\underline{m}, e)_m = true\} : L \Rightarrow G$*.*
- *For all sets of matches* $\underline{m} : L_1 \Rightarrow G_1$ *and* $\underline{h} : L_2 \Rightarrow G_2$*:*
  $Join(\underline{m}, \underline{h}) = \{m \bowtie p \mid m \in \underline{m} \wedge p \in \underline{h} \wedge m \sim p\} : L_1 \cup L_2 \Rightarrow G_1 \cup G_2$*.*
- *For every set of matches* $\underline{m} : L \Rightarrow G$ *from an open PGP graph* L *to a PGP graph* G*. For all* x*, a node or an edge, in* L*,* p *a property in* $Prop \cup \mathcal{V}_{prp}(L)$ *and* v *a value in* Val*:*
  $Set(\underline{m}, e, x, p) : L \Rightarrow \cup_{m \in \underline{m}} G[m(x).m(p) \leftarrow ev(\underline{m}, e)_m]$

– *For every set of matches* $\underline{m} : L \Rightarrow G$ *from an open PGP graph* L *to a PGP graph* G *and every open PGP graph* R *such that* $\mathcal{V}_{pth}(R) \subseteq \mathcal{V}_{pth}(L)$, $\mathcal{V}_{lab}(R) \subseteq \mathcal{V}_{lab}(L)$ *and* $\mathcal{V}_{prp}(R) \subseteq \mathcal{V}_{prp}(L)$*:*
$Build(\underline{m}, R) = \{Build(m, R) \mid m \in \underline{m}\} : R \Rightarrow G \cup \cup_{m \in \underline{m}} Build(m, R)(R)$.
Let $\cong$ *be a graph equivalence on* $G \cup \cup_{m \in \underline{m}} Build(m, R)(R)$*. Then*
$Buildeq(\underline{m}, R, \cong) = Build(m, R)_{\cong} : R \Rightarrow [G \cup \cup_{m \in \underline{m}} Build(m, R)(R)]_{\cong}$.

3.3 Patterns and CONSTRUCT Queries

Patterns are the main building blocks of queries. They can be nested. The semantics of a pattern $Patt$ is defined as a set of matches. The source graph of such matches is a graph called *scope graph* of pattern $Patt$ and denoted by $[Patt]$.

Definition 16 (syntax of patterns). Patterns $Patt$ *and their* scope graphs $[Patt]$ *are defined recursively as follows.*

– *The symbol* $\square$ *is a pattern, called the* empty pattern*, and* $[\square]$ *is the empty graph* $\emptyset$.
– *If* L *is an open property graph with paths then* $Patt = BASIC(L)$ *is a pattern, called a* basic pattern*, and* $[Patt] = L$.
– *If* P_1 *is a pattern and* e *a boolean expression such that* $\mathcal{V}(e) \subseteq \mathcal{V}([P_1])$ *then* $Patt = (P_1\ FILTER\ e)$ *is a pattern and* $[Patt] = [P_1]$.
– *If* P_1 *and* P_2 *are patterns then* $Patt = P_1\ JOIN\ P_2$ *is a pattern and* $[Patt] = [P_1] \cup [P_2]$.
– *If* P_1 *is a pattern,* e *a value expression such that* $\mathcal{V}(e) \subseteq \mathcal{V}([P_1])$*,* x *a node in* $[P_1]$ *and* p *is a property (in* $Prop \cup \mathcal{V}_{prp}([P_1])$*) then* $Patt = (P_1\ SET\ x.p = e)$ *is a pattern and* $[Patt] = [P_1]$.
– *If* P_1 *is a pattern,* R *an open PGP such that* $\mathcal{V}_{pth}(R) \subseteq \mathcal{V}_{pth}(P_1)$, $\mathcal{V}_{lab}(R) \subseteq \mathcal{V}_{lab}(P_1)$ *and* $\mathcal{V}_{prp}(R) \subseteq \mathcal{V}_{prp}(P_1)$*, that is the path variables, the label variables and the property variables appearing in* R *also occur in* P_1*,* $x \in N_R$ *and* p *is either a property (*$\in Prop$*) or a star* $\star$*, then*
$Patt = (P_1\ BUILD\ R\ [GROUP\ x.p]^*)$ *is a pattern and* $[Patt] = R$.

Definition 17 (evaluation of patterns, set of solutions). *The* set of solutions *or the* value *of a pattern* $Patt$ *over a PGP,* G*, is a set of matches* $[\![Patt]\!]_G : [Patt] \Rightarrow G^{(Patt)}$ *from the scope graph* $[Patt]$ *of* $Patt$ *to a graph* $G^{(Patt)}$ *that contains* G*. This value* $[\![Patt]\!]_G : [Patt] \Rightarrow G^{(Patt)}$ *is defined inductively as follows:*

– $[\![\square]\!]_G = \varnothing_G : \emptyset \Rightarrow G$.
– $[\![BASIC(L)]\!]_G = Match(L, G) : L \Rightarrow G$.
– $[\![P_1\ FILTER\ e]\!]_G = Filter([\![P_1]\!]_G, e) : [P_1] \Rightarrow G^{(P_1)}$.
– $[\![P_1\ JOIN\ P_2]\!]_G = Join([\![P_1]\!]_G, [\![P_2]\!]_{G^{(P_1)}}) : [P_1] \cup [P_2] \Rightarrow G^{(P_1)^{(P_2)}}$.
– $[\![P_1\ SET\ x.p = e]\!]_G =$
$Set([\![P_1]\!]_G, e, x, p) : [P_1] \Rightarrow \cup_{m \in [\![P_1]\!]_G} G^{(P_1)}[m(x).m(p) \leftarrow ev([\![P_1]\!]_G, e)_m]$.

– $[\![P_1\ BUILD\ R\ [GROUP\ x.p]^*]\!]_G =$
$Buildeq([\![P_1]\!]_G, R, \cong_{[x.p]^*}) : R \Rightarrow [G^{(P_1)} \cup (Build([\![P_1]\!]_G, R)]_{\cong_{[x.p]^*}}$.

The BUILD clause is key in the construction of new data. The evaluation of a pattern of the form P_1 BUILD R with respect to a graph G, generates a new copy of graph R for every match m_i in the set of matches $[\![P_1]\!]_G$. However, there are situations where the generation of new copies of R may go beyond the desired output data mainly due to duplication of nodes(cf. the institution nodes in Example 7). To overcome such a situation, we provide a declarative way to specify graph congruences, the role of which is to make equal some of the nodes which have been generated separately by the BUILD clause. For that we use the GROUP clause, inspired from [1], to specify graph congruences. In a pattern P_1 BUILD R [GROUP $x.p$]*, the optional clause [GROUP $x.p$]* provides the possibility to equate or merge some nodes, image of given nodes x in R, by using an induced graph equivalence, $\cong_{[x.p]^*}$. Two different nodes u and v are $\cong_{[x.p]^*}$ equivalent if u and v are both images of one node x in R via two different matches m_1 and m_2 in $G^{(P_1)}$, say $m_1(x) = u$ and $m_2(x) = v$, such that u and v share a common value for property p. That is, there exists a value c with (p, c) is in $\pi(m_1(x)) \cap \pi(m_2(x))$. Whenever p is a star, all nodes images of a node x in R are equivalent regardless the values associated to the properties. Recall that the equivalence $\cong_{[x.p]^*}$ is such that x represents a node and p a property. Hence, $\cong_{[x.p]^*}$ does not equate edges, i.e., two edges e_1 and e_2 are $\cong_{[x.p]^*}$ equivalent if and only if e_1 and e_2 are identical ($e_1 = e_2$). We do not allow x to be an edge because otherwise the generated equivalence, $\cong_{[x.p]^*}$, would not be a graph equivalence in general (Definition 5), that is, two edges may be equivalent but not their sources or targets.

Example 4. We consider pattern BASIC($L1$) where $L1$ is the graph defined in Example 2 and define two new patterns $L2$ and $L3$. The role of $L2$ is to select only matches that maps different authors, that is the matches which satisfy the condition $x <> y$. These matches $[\![L2]\!]_{G1}$ are $\{m_1, m_2, m_3, m_4\}$ seen in Example 2. Pattern $L3$ constructs the relationship *coauthor* deduced from the evaluation of pattern $L2$ by building graph `R3 = (x) -[u:coauthor]-> (y)`.

```
L2 = BASIC(L1)  FILTER  x <> y
L3 = L2  BUILD  R3
```

The evaluation of pattern L3, $[\![L3]\!]_{G1} : R3 \Rightarrow G3$, transforms graph $G1$ into graph $G3$, depicted in Fig. 2, by adding the coauthor edges.

Example 5. To illustrate the use of JOIN operation to build a pattern, we reconsider pattern $L2$ again. $L2$ can be reformulated as a join of two basic patterns before the filter clause.

```
L2bis = (BASIC( (x:person)-[x1:author]->(z:paper)) JOIN
         BASIC((y:person)-[y1:author]->(z:paper)))
         FILTER  x <> y
```

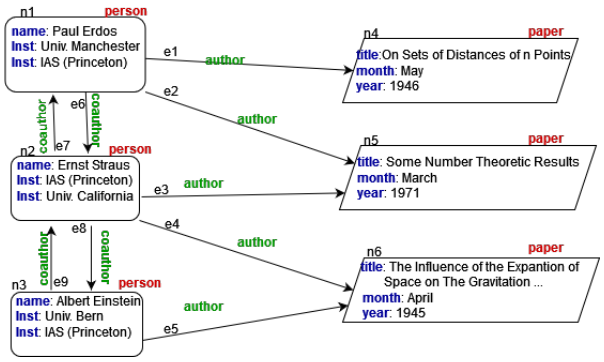

Fig. 2. Property Graph $G3$, target of $[\![L3]\!]_{G1}$

Example 6. We illustrate here the use of aggregation and path variables in a pattern. We define pattern $L4$ which consists of a path variable w. The source of the path variable w is a node of label person, `(x:person)` and its target is the node corresponding to the author P. Erdős, `(y: person)` where `y.name = Paul Erdos`. Two properties are required from path w: (i) to be a trail (no edge can occur twice in w). This condition makes finite the possible paths in the considered graph to match with. (ii) to be shortest. An additional condition over w requires that all edges composing w have labels `coauthor`, written as $w : coauthor^k$. Then, for each author, the length of such path w provides the Erdős number which states the collaboration distance between an author and P. Erdős. We write pattern L4 as follows:

```
L4 =  BASIC((x:person) -[w:<Shortest, Trail> coauthor^k] (y:person))
      WHERE k>=0,  y.name = Paul Erdos
      SET x.Erdos number = length(w)
```

The evaluation of $[\![L4]\!]_{G3}$ is obvious. Its target graph $G4$ is illustrated in Fig. 3 where properties *Erdos number* have been filled.

Example 7. We illustrate here the use of the `GROUP` clause. We define pattern $L5$ which builds, for each author, new nodes corresponding to the institutions the author worked at. Such nodes should be grouped by their names. That is, two nodes representing the same institution should be merged. An edge of label `worked-at` is added between authors and their institutions. Pattern $L5$ can be defined as follows:

```
L5 = BASIC((x:person)) BUILD (x) -[v:worked-at]->(i:institution)
                       SET i.name = x.inst
                       GROUP i.name
```

The evaluation of pattern $L5$ over graph $G4$, $[\![L5]\!]_{G4}$, is a set of matches whose target graph $G5$ is depicted in Fig 4. Notice that node $n8$ has been generated

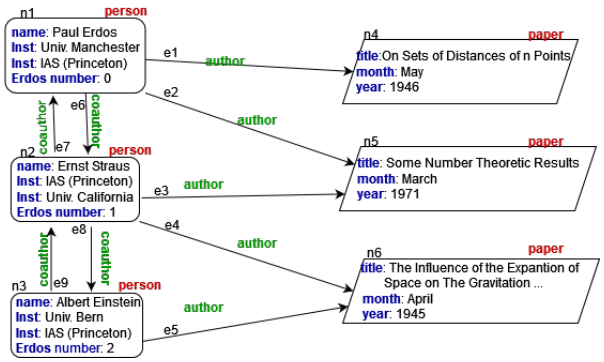

Fig. 3. Graph $G4$ illustrating the computation of Erdős numbers

only once instead of three thanks to the clause `GROUP i.name` which grouped all nodes with the same institution name. The three authors of the running example did work at `IAS (Princeton)`.

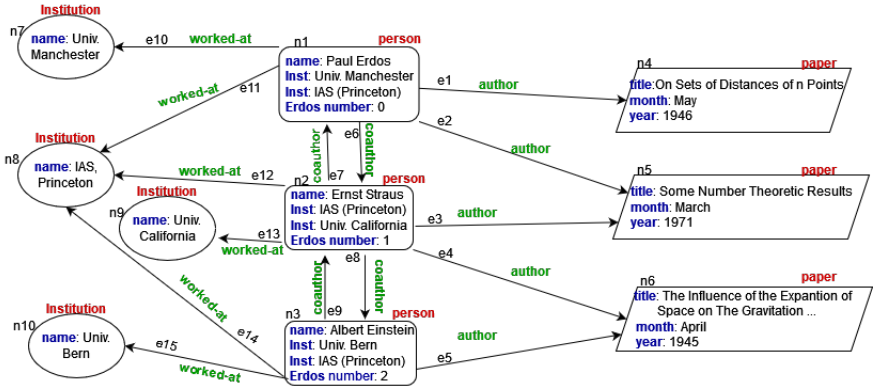

Fig. 4. Graph $G5$ with new institution nodes grouped by names

Below we define the `CONSTRUCT` queries.

Definition 18 (CONSTRUCT Queries). *Let L be a pattern, R an open PGP, x a node in R and p a property. A CONSTRUCT query, Q,is of the form*

$$CONSTRUCT\ R\ [GROUP\ x.p]^*\ WHERE\ L$$

The GROUP clause is optional. The result *of a 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 $[\![P]\!]_G$ where P is the pattern $P = L\ BUILD\ R\ [GROUP\ x.p]^*$.*

From the definition of CONSTRUCT queries, we notice that a query specifies only the part (graph R) of the target graph to be displayed. Hence, the main important syntactic notion is that of patterns. In the following section, we investigate an operational semantics for patterns and queries.

4 A Rule-based Operational Semantics

In this section we use rewriting techniques to define an operational semantics to solve patterns and queries. In classical declarative programming languages, goal solving procedures are based on some well known techniques such as narrowing [3] or resolution [9]. Solving a goal g_0 consists in developing derivations of the form

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

where g_{n+1} is a "terminal" goal such as the empty clause, unifiable equations or the constant *true*. 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 pattern or a query and the underlying program is a graph (i.e., a database) augmented by a set of rewriting rules defining the behavior of two functions *Solve* (for patterns) and $Solve_Q$ (for queries). Due to the use of aggregation operators in the patterns, one has to compute all solutions simultaneously and thus handle a set of substitutions or matches instead of one substitution at each step as in classical declarative languages. In order to have an easy way to handle such sets of matches, we introduce below the notion of configuration.

Definition 19 (configuration). *Let $\underline{m} : L \Rightarrow G$ be a set of matches from L to G and P a pattern. A configuration is denoted using a mixfix notation as a pair $[P, \underline{m} : L \Rightarrow G]$ or simply $[P, \underline{m}]$. An* initial *configuration is a configuration of the form $[P, \underline{i}_G : \emptyset \Rightarrow G]$ where $\underline{i}_G = Match(\emptyset, G)$ is the set with one unique element that is the inclusion of the empty graph, $\emptyset$, into G. A* terminal *configuration is a configuration of the form $[\Box, \underline{m} : L \Rightarrow G]$.*

A configuration $[P, \underline{m} : L \Rightarrow G]$ represents a state where P is a pattern to solve with respect to the current database G. G is the target of the current set of matches $\underline{m} : L \Rightarrow G$. Finding solutions of a pattern P over a graph G consists in starting from the term $Solve([P, \underline{i}_G : \emptyset \Rightarrow G])$ which applies the function *Solve* to an initial configuration by performing appropriate rewriting rules to transform configurations until reaching a terminal configuration of the form $[\Box, \underline{m} : L \Rightarrow G']$ where $\underline{m} : L \Rightarrow G'$ represents the expected set of matches (solutions) of P over G and where G' is the graph obtained after solving the pattern P over G. Notice that G' contains G but they are not necessarily equal.

In Fig. 5, we provide a rewriting system, $\mathcal{R}_{gq}^{PG}$, which defines the function *Solve*. This function is defined by structural induction on the first component of configurations, i.e., on patterns. The second argument of configurations, i.e., the sets of matches, in the left-hand sides defining the function *Solve* are always variables of the form $\underline{m} : L \Rightarrow G$ or simply $\underline{m}$ and thus can be handled easily

in the pattern-matching process of the left-hand sides of the proposed rules (no need to higher-order pattern-matching nor unification). In the rules of $\mathcal{R}_{gq}^{PG}$, the variable P ranges over *patterns* while variables L, G and R are ranging over PGP *graphs* and $\emptyset$ is the constant denoting the empty graph. Symbol e is a variable over value expressions and x (resp. p) is a variable ranging over nodes (resp. over properties) while $\underline{m}$ and $\underline{h}$ are variables ranging over sets of matches. Some constraints of the rules use operations already introduced in Definition 15, such as *Match*, *Set*, *Filter*, *Buildeq*.

Fig. 5. $\mathcal{R}_{gq}^{PG}$: Rewriting rules for patterns

$$
\begin{array}{lll}
r_0 & : & \mathit{Solve}([\,\Box,\,\underline{m}\,]) \to [\,\Box,\,\varnothing_G : \emptyset \Rightarrow G\,] \\
\\
r_1 & : & \mathit{Solve}([\,\mathrm{BASIC}(L),\,\underline{m}\,]) \to [\,\Box,\,\underline{h} : L \Rightarrow G\,] \\
 & & \quad \text{where } \underline{h} = \mathit{Match}(L, G) \\
\\
r_2 & : & \mathit{Solve}([\,P \text{ SET } x.p = e,\,\underline{m}\,]) \to \mathit{Solve}_{ST}\,(\mathit{Solve}([\,P,\,\underline{m}\,]), e, x, p) \\
r_3 & : & \mathit{Solve}_{ST}\,([\,\Box,\,\underline{m}\,], e, x, p) \to [\,\Box,\,\underline{h}\,] \\
 & & \quad \text{where } \underline{h} = \mathit{Set}(\underline{m}, e, x, p) \\
\\
r_4 & : & \mathit{Solve}([\,P \text{ FILTER } e,\,\underline{m}\,]) \to \mathit{Solve}_{FR}\,(\mathit{Solve}([\,P,\,\underline{m}\,]), e) \\
r_5 & : & \mathit{Solve}_{FR}\,([\,\Box,\,\underline{m}\,], e) \to [\,\Box,\,\underline{h}\,] \\
 & & \quad \text{where } \underline{h} = \mathit{Filter}(\underline{m}, e) \\
\\
r_6 & : \mathit{Solve}([\,P \text{ BUILD } R\,[\text{ GROUP } x.p]^*,\,\underline{m}\,]) \to \mathit{Solve}_{BU}(\mathit{Solve}([\,P,\,\underline{m}\,]), R, [x.p]^*) \\
r_7 & : & \mathit{Solve}_{BU}\,([\,\Box,\,\underline{m}\,], R, \mathit{cong}) \to [\,\Box,\,\underline{h}\,] \\
 & & \quad \text{where } \underline{h} = \mathit{Buildeq}(\underline{m}, R, \cong_{\mathit{cong}}) \\
\\
r_8 & : & \mathit{Solve}([\,P_1 \text{ JOIN } P_2,\,\underline{m}\,]) \to \mathit{Solve}_{JL}\,(\mathit{Solve}([\,P_1,\,\underline{m}\,]), P_2) \\
r_9 & : & \mathit{Solve}_{JL}\,([\,\Box,\,\underline{m}\,], P) \to \mathit{Solve}_{JR}\,(\underline{m}, \mathit{Solve}([\,P,\,\underline{m}\,])) \\
r_{10} & : & \mathit{Solve}_{JR}\,(\underline{m}, [\,\Box,\,\underline{m}'\,]) \to [\,\Box,\,\underline{h}\,] \\
 & & \quad \text{where } \underline{h} = \mathit{Join}(\underline{m}, \underline{m}')
\end{array}
$$

In the sequel, we write $\mathcal{P}_{gq}(\mathcal{X})$ for the term algebra over the set of variables $\mathcal{X}$ generated by the operations occurring in the rewriting system $\mathcal{R}_{gq}^{PG}$.

Rule r_0 considers the degenerated case of the empty pattern $\Box$. In this case there is no solution and the empty set of matches $\varnothing_G$ is computed.

Rule r_1 is key in this system because it considers basic patterns of the form BASIC(L) where L is a PGP graph with possible variables to be instantiated. In this case $\mathit{Solve}([\,\mathrm{BASIC}(L),\,\underline{m}\,])$ consists in finding all matches from L to G. These matches instantiate variables in L thanks to the constraint $\underline{h} = \mathit{Match}(L, G)$ of rule r_1. This variable instantiation process is close to the narrowing-based procedures [3]. In this paper, we do not need all the power of narrowing procedures because manipulated data are mostly flat (mainly con-

stants and variables). Thus the unification process used at every step in the narrowing relation is beyond our needs. On the other hand, the classical rewriting relation induced by the above rewriting system is not enough since variables occurring in patterns have to be instantiated and such an instantiation cannot be done by simply rewriting the initial term $Solve([P, \underline{i}_G : \emptyset \Rightarrow G])$. Therefore, we propose hereafter a relation induced by the above rewriting system that we call *gq-narrowing*. Before defining this relation, we recall some notations about first-order terms. Readers not familiar with such notations may consult, e.g., [4].

Definition 20 (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 is 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 $t[s]_p$. $t{\downarrow_{gq}}$ is the term obtained from t after evaluation of all expressions (i.e., operations such as Filter, Match, Buileq etc.). 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 $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.*

Definition 21 (gq-narrowing $\rightsquigarrow$). *The rewriting system $\mathcal{R}_{gq}^{PG}$ 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 $\mathcal{R}_{gq}^{PG}$, 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 above of term $t' = t[\sigma(rhs){\downarrow_{gq}}]_u$, 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 (i) the fact that the unification process is pushed back into rules' constraints and (ii) the fact that aggregations require local instantiations only. Therefore there is no need to propagate variable instantiations. Actually, If we consider again rule r_1, t' would be of the form $t' = t[\square, (Match(\sigma(L), G) : \sigma(L) \Rightarrow G){\downarrow_{gq}}]_u$ where the evaluation of the *Match* operation instantiates variables occurring in the pattern (or goal to solve) $BASIC(\sigma(L))$ just like classical narrowing procedures when rules such as $f(t_1, \ldots, t_n) \to r$ are transformed or flattened into $f(x_1, \ldots, x_n) \to r \mid \bigwedge_1^n Unif(x_i, t_i)$.

Definition 22 (gq-narrowing derivations). *Let G be a graph, P a pattern and $\underline{m}$ a set of matches. The evaluation of P over G consists in computing gq-narrowing derivations of the form:*

$$Solve([P, \underline{i}_G : \emptyset \Rightarrow G]) \rightsquigarrow^* [\square, \underline{m}]$$

Notice that $Solve([P, \underline{i}_G : \emptyset \Rightarrow G])$ is intended to find both the solutions (matches) of pattern P over graph G and the graph G' obtained after transforming graph G along the evaluation of the sub-patterns of P.

Example 8. We consider pattern $L3$ as defined in Example 4. $L3 = L2$ BUILD $R3$ where $L2 = \text{BASIC}(L1)$ FILTER $x <> y$.

The expected gq-narrowing derivation over G_1 is as follows:

$$
\begin{aligned}
&Solve(\,[\,L3,\,\underline{i}_{G1}\,]\,)\\
&\quad\rightsquigarrow_{r_6}\ Solve_{BU}(Solve(\,[\,L2,\,\underline{i}_{G1}\,]\,), R3, id\,)\\
&\quad\rightsquigarrow_{r_4}\ Solve_{BU}(Solve_{FR}\,(Solve(\,[\,BASIC(L1),\,\underline{i}_{G1}\,]\,), x <> y), R3, id\,)\\
&\quad\rightsquigarrow_{r_1}\ Solve_{BU}(Solve_{FR}\,(\,[\,\square,\ Match(L1, G1)\,]\,, x <> y), R3, id\,)\\
&\quad\rightsquigarrow_{r_5}\ Solve_{BU}([\,\square,\ Filter(Match(L1, G1), x <> y)\,]\,, R3, id\,)\\
&\quad\rightsquigarrow_{r_7}\ [\,\square,\ Buildeq(Filter(Match(L1, G1), x <> y), R3, id)\,]
\end{aligned}
$$

The expression $Buildeq(Filter(Match(L1, G1), x <> y), R3, id)$ evaluates to a set of matches whose codomain is exactly the graph displayed in Fig.2. Notice that when the GROUP clause is empty, the equivalence on nodes is the identity (reflexivity) denoted by the symbol *id* in the third argument of the *Buildeq* operator.

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 $i \in [0..n]$, there exists at most one position u_i in t_i such that t_i can be gq-narrowed into t_{i+1}.*

Theorem 1 (soundness). *Let G be a graph, P a pattern and $\underline{m}$ a set of matches such that $Solve([P, \underline{i}_G]) \rightsquigarrow^* [\square, \underline{m}]$. 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 $Solve([P, \underline{i}_{G_1}] \rightsquigarrow^* [\square, \underline{m}]$ and a match $m : X' \Rightarrow G'_2$ in $\underline{m}$ such that m and h are equal up to variable renaming.*

In Fig. 6, we enrich the rewriting system $\mathcal{R}_{gq}^{PG}$ by means of two additional rules which tackle CONSTRUCT queries. the *result* of a CONSTRUCT query Q over a graph G is a graph $Result_C(Q, G)$. The specification of the display function $Print_C$ is out of the scope of the present paper. The Soundness and completeness of the calculus with respect to CONSTRUCT queries are direct consequences of Theorems 1 and 2.

5 Conclusion and Related Work

We proposed a symbolic procedure based on rewriting techniques to solve patterns or CONSTRUCT graph queries for a core graph query language. The

Fig. 6. $\mathcal{R}_{gq}^{PG}$ (continued): Rewriting rules for queries

r_{11} : $Solve_Q$(CONSTRUCT R [GROUP $x.p$]* WHERE P, G) → $Display_C(R, Solve([\, P$ BUILD R [GROUP $x.p]^*, \underline{i}_G\,]))$
r_{12} : $Display_C(R, [\square, \underline{m}]) \rightarrow Print_C(R, \underline{m})$

considered data graphs are represented by the so-called property graphs. The proposed procedure is deterministic, sound and complete. By the nature of the considered graphs, our procedure differs from the one presented for RDF graphs in [7]. The operators on property graphs feature new needs such as SET or GROUP clauses in addition to the way paths play an important role in queries.

The notion of *pattern* present in this paper is close to the syntactic notions of *clauses* in [8] or *graph patterns* in [2]. For such syntactic notions, some authors associate as semantics sets of variables bindings (tables) as in [8, 10] or simply graphs as in [1]. 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 first work on formal semantics of graph queries based on category theory [6]. 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 [1].

References

1. Angles, R., Arenas, M., Barceló, P., Boncz, P.A., Fletcher, G.H.L., Gutiérrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J.F., van Rest, O., Voigt, H.: G-CORE: A core for future graph query languages. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. pp. 1421–1432. ACM (2018). https://doi.org/10.1145/3183713.3190654
2. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1–68:40 (2017). https://doi.org/10.1145/3104031
3. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM **47**(4), 776–822 (2000). https://doi.org/10.1145/347476.347484
4. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press (1998)
5. Duval, D., Echahed, R., Prost, F.: All you need is CONSTRUCT. CoRR **abs/2010.00843** (2020), https://arxiv.org/abs/2010.00843
6. Duval, D., Echahed, R., Prost, F.: Querying RDF databases with sub-constructs. In: Kutsia, T. (ed.) Proceedings of the 9th International Symposium on Symbolic Computation in Software Science, SCSS 2021, Hagenberg, Austria, September 8-10, 2021. EPTCS, vol. 342, pp. 49–64 (2021). https://doi.org/10.4204/EPTCS.342.5
7. Duval, D., Echahed, R., Prost, F.: A rule-based procedure for graph query solving. In: 16th International Conference on Graph Transformation, ICGT 2023, Held as Part of STAF 2023, Leicester, UK, July 19-20, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13961, pp. 163–183. Springer (2023). https://doi.org/10.1007/978-3-031-36709-0_9

8. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: SIGMOD Conference. pp. 1433–1445. ACM (2018)
9. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987). https://doi.org/10.1007/978-3-642-83189-8
10. Pérez, J., Arenas, M., Gutiérrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16:1–16:45 (2009). https://doi.org/10.1145/1567274.1567278
11. SPARQL 1.1 Query Language. W3C Recommendation (march 2013), https://www.w3.org/TR/sparql11-query/
12. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (February 2014), https://www.w3.org/TR/rdf11-concepts/