Typing Query Languages for Data Graphs

Dario Colazzo #1, Carlo Sartiani *2

# LAMSADE, Université Paris Dauphine
Place du Maréchal de Lattre de Tassigny, 75 775 Paris Cedex 16, France
1 dario.colazzo@dauphine.fr
* DIMIE, Università della Basilicata
Via dell’Ateneo Lucano 10 - Potenza - Italy
2 sartiani@gmail.com

Abstract—Graph query languages are essentially untyped. The lack of type information greatly limits the optimization opportunities for query engines and makes application development more complex. In this paper we discuss a simple, yet expressive, schema language for edge-labelled data graphs. This schema language is then used to define a query type inference approach with good precision properties.

I. INTRODUCTION

In the last few years graph databases gained more and more relevance in application areas such as the Semantic Web, social networks, bioinformatics, network traffic analysis, and crime detection. This led to the definition of many query formalisms for graph databases, like, for instance, regular path queries (RPQs [1]), nested regular expressions (NREs [2]), conjunctive regular path queries (CRPQs [3]), GXPath [4], and their derivatives. All these languages are based on the idea of specifying regular expressions describing paths in the input graph, and can be considered, to some extent, a generalization of existing path query languages for semistructured data (see XPath [5], for instance).

Graph query languages are essentially untyped. This means that one cannot statically infer the structure of query results (type inference), check if the results satisfy a given schema (type-checking), and verify if the query results would always be empty (query correctness). Furthermore, the lack of type information greatly limits the optimization opportunities for query engines and makes application development more complex.

Our Contribution: In this paper we present preliminary results about a simple, yet expressive, schema language for edge-labelled data graphs. A schema is formed by a collection of schema elements, each one describing the set of incoming and outgoing edges of a class of graph nodes; edges are specified through regular expressions. We show that restrictions on regular expressions describing edges are needed in order to ensure that the semantics of graph schemas is well defined. The resulting schema language is named GTD (Graph Type Definition) and can be viewed as a generalization of DTDs [6] to data graphs. The proposed language is a first step towards the definition and analysis of more powerful schema languages for data graphs.

In the second part of the paper we illustrate how GTDs can be used to define a type inference system for a large fragment of GXPath. Given an input query q, a graph G, and a schema S for G, this system infers not only type information about the node pairs returned by q, but it also extracts detailed information about the paths connecting these nodes.

II. PRELIMINARY DEFINITIONS

A. Data Model and Type Language

Following [4], we model a data graph as an edge-labelled graph, as shown below.

Definition II.1 (Data Graph) Given a finite alphabet \( \Sigma \) and a (possibly) infinite value domain \( \mathcal{D} \), a data graph \( G \) over \( \Sigma \) and \( \mathcal{D} \) is a triple \( G = (V,E,\rho) \), where:

- \( V \) is a finite set of nodes;
- \( E \subseteq V \times \Sigma \times V \) is a set of labelled, directed edges \( (v_i,a,v_j) \);
- \( \rho : V \rightarrow \mathcal{D} \) is a mapping from nodes to values.

Given a node \( v \), we indicate with \( \text{in}(v) \) and \( \text{out}(v) \) the set of incoming and outgoing edges, respectively. Formally:

- \( \text{in}(v) = \{ (v',a,v) \in E \mid v' \in V \land a \in \Sigma \} \);
- \( \text{out}(v) = \{ (v,a,v') \in E \mid v' \in V \land a \in \Sigma \} \).

We assume that sequences of outgoing (incoming) edges of a node are unordered, as it is often the case in graph databases. Given a set of edges \( S_E \subseteq E \), we will indicate with \( \pi(S_E) \) the unordered concatenation of the labels of the edges in \( S_E \).

This data model is general enough to capture many practical use graphs, ranging from RDF data to social network graphs, as shown by the following example.

Example II.2 Consider the graph shown in Figure 1. This graph contains bibliographic information coming from a fragment of the RDF representation of the DBLP repository [7]. As in [4], we indicate the value of a node inside its graphical representation, and use RDF properties to label edges. In this work we propose a schema language for data graphs that associates to each schema element a pair of regular expressions describing sequences of labels of the incoming and outgoing edges of each node. Regular expressions obey the following grammar:

\[
T ::= \epsilon \mid a \mid T + T \mid T \& T \mid T^* 
\]

where \( \epsilon \) denotes the empty sequence, \( a \) is a symbol in \( \Sigma \), + and \& denote, respectively, union and unordered concatenation,
and * is the Kleene star. As expected, unordered concatenation & is commutative, associative and has ε as neutral element. In particular, the expression $T_1 \& \ldots \& T_n$ is equivalent to all of its possible permutations. In the following we will also use $T^+$ and $T^?$ as abbreviations for $T^* \cdot T$ and $T + \epsilon$.

The semantics of regular expressions is denoted as $L(-)$, denoting the minimal function satisfying the following equations, where $L_1 \& L_2$ denotes unordered language concatenation and is defined in the obvious way, while for any $i \in \mathbb{N}$, $L^i = L \& L^{i-1}$ with $L^0 = \{\epsilon\}$:

$$
\begin{align*}
L(\epsilon) &= \{\epsilon\} \\
L(a) &= \{a\} \\
L(T_1 + T_2) &= L(T_1) \cup L(T_2) \\
L(T_1 \& T_2) &= L(T_1) \& L(T_2) \\
L(T^*) &= \bigcup_{i \geq 0} L(T)^i
\end{align*}
$$

**B. Schema Language**

Regular expressions are the building blocks of our schema language.

**Definition II.3** Given a regular expression $r$ over Σ, $\text{sym}(r)$ is the set of symbols in Σ appearing in $r$.

**Definition II.4 (Graph Schema Element)** Given a finite alphabet Σ, a schema element $e$ over Σ is a pair $(e\.in, e\.out)$, where $e\.in$ and $e\.out$ are regular expressions over Σ.

The semantics of a schema element $e$ is defined as follows.

$$
[e] = \{v \mid \pi(in(v)) \in L(e\.in) \land \pi(out(v)) \in L(e\.out)\}
$$

A schema element, then, specifies constraints on the incoming and outgoing edges of a node. Consider, for instance, the following schema element: $e = (a \& b \& c + d) \& e^*$). This element describes graph nodes having an incoming $a$-edge, an incoming $b$-edge, as well as an incoming edge labelled with $c$ or $d$; these nodes must also have an outgoing $c$-edge together with zero or more outgoing $h$-edges.

**Definition II.5 (Graph Schemas)** A graph schema $S$ is a finite set of schema elements $\{e_i\}_{i=0}^n$ such that:

1. $\forall i \in [0..n]. \forall l \in \text{sym}(e_i\.in). \exists j \in [0..n]. l \in \text{sym}(e_j\.out)$;
2. $\forall i \in [0..n]. \forall l \in \text{sym}(e_i\.out). \exists j \in [0..n]. l \in \text{sym}(e_j\.in)$;
3. $\forall i, j \in [0..n] : (e_i\.in \cap e_j\.in = \emptyset \lor e_i\.out \cap e_j\.out = \emptyset)$.

Conditions 1 and 2 above are necessary to ensure that the schema cannot define graphs with dangling edges: any symbol used in an outgoing edge must also be used to label an incoming edge, and vice versa. As we will see later, these conditions are not sufficient to imply non-emptiness. Condition 3 guarantees the uniqueness of node typing: a graph node can be typed by at most one schema element.

Schema semantics is defined as follows.

**Definition II.6 (Graph Schema Semantics)** A data graph $G = (V, E, \rho)$ over Σ and $\mathcal{D}$ is described by a graph schema $S$ ($G \in \llbracket S \rrbracket$) if and only for each $v \in V$ there exists $e_i \in S$ such that $v \in \llbracket e_i \rrbracket$.

**Example II.7** Consider again the graph of Example II.2. This graph can be typed by the schema $S = \{e_1, e_2, e_3, e_4, e_5\}$, where:

$$
e_1 = (e, (\text{journal} \lor \text{partOf}) \& (\text{creator})^+) \\
e_2 = (\text{journal}, e) \\
e_3 = (\text{partOf} \& e, \text{series}) \\
e_4 = (\text{series}, e) \\
e_5 = (\text{creator}, e)
$$

A graph schema may be empty, and it could be difficult for the user to figure out whether the schema she has defined is empty. For a simple schema like the following one, emptiness can be easily detected.

**Example II.8** Consider the graph schema $S = \{e_1, e_2\}$, where:

$$
e_1 = (e, a \& b \& c \& c) \\
e_2 = (a \& b \& c, e)
$$

This schema satisfies conditions 1) and 2) of Definition II.5. However, it is empty as cardinality constraints expressed by regular expressions of incoming and outgoing edges are incompatible.

For some schemata, checking compatibility between incoming and outgoing edges can be far from being obvious, as happens for the following one.

**Example II.9** Consider the graph schema $S = \{e_1, e_2, e_3\}$, where:

$$
e_1 = (e, a \& b \& c \& c \& c) \\
e_2 = (a \& b \& c, e) \\
e_3 = (c, e)
$$

In this schema each $e_3$ node produces 4 outgoing $c$-edges, that are consumed by $e_2$ and $e_3$ nodes. This schema is not empty, as it is possible to build a well-formed graph comprising 2 $e_1$ nodes, 2 $e_2$ nodes, and 3 $e_3$ nodes.

To overcome this problem we will devise a characterization of empty schemata, as well as a correct and complete technique for checking if a graph schema is empty. The basic idea behind this characterization is to associate to each schema element a variable and to build, for each symbol, a polynomial equation describing the produced and consumed edges labelled with that symbol: in particular, each symbol equation contains the variables of the schema elements producing or consuming
edges labelled with that symbol; the coefficient of each variable describes the number of produced or consumed edges. The result is an homogeneous system which has a non-zero integer solution if and only if the schema is not empty. The following example illustrates our idea.

**Example II.10** Consider again the schema of Example II.8. This empty schema consists of two schema elements (e₁ and e₂) to which we can associate variables x and y. Regular expressions in the schema use three different symbols (a, b, and c), so we have to define the following three linear equations:

\[ a. \quad x - y = 0 \]
\[ b. \quad x - y = 0 \]
\[ c. \quad 2x - y = 0 \]

In the first equation variable x has coefficient 1, as e₁ produces an a-edge, while variable y has coefficient −1 since e₂ consumes an a-edge. As it can be easily seen, the only solution of this system is (0,0,0).

Consider now the schema of Example II.9. As illustrated before, this schema is not empty and comprises three schema elements (e₁, e₂, and e₃) to which we can associate variables x, y, and z. As for the previous example, we have three distinct symbols in the schema, so we can define a system with the following linear equations:

\[ a. \quad x - y = 0 \]
\[ b. \quad x - y = 0 \]
\[ c. \quad 4x - y - 2z = 0 \]

It easy to see that (2,2,3) is a solution for this system. This means that it is possible to build a graph with 2 e₁ vertices, 2 e₂ vertices, and 3 e₃ vertices.

The definition of a sound and complete system construction technique generalising the just illustrated idea, as well as the study of its complexity, is the subject of our current research activities.

### III. Query Language

GXPath, proposed by Libkin et al. in [4], is a graph query language extending RPQs and NREs with new features inspired by those present in XPath. We focus here on the navigational fragment with intersection and complement, described by the following grammar.

\[ \alpha \ ::= \epsilon \mid \alpha \mid a^- \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha^m \cdot n \mid \alpha \cap \alpha \mid [\alpha] \mid \overline{\alpha} \]

where _ matches any symbol, a⁻ denotes a backward navigation, + and · are the standard union and ordered concatenation operators, \( \alpha^m \cdot n \), where \( m \in \mathbb{N} \cup \{0\} \) denotes the repetition from \( m \) to \( n \) times, \( \alpha_1 \cap \alpha_2 \) is the intersection of \( \alpha_1 \) and \( \alpha_2 \), \([\alpha] \) denotes a nested condition (the syntax is borrowed from XPath), and \( \overline{\alpha} \) is the complement of \( \alpha \).

Given a graph \( G = (V, E, \rho) \), the semantics of GXPath can be defined as follows.

\[ [e]_G = \{(u, v) \mid u \in V\} \]
\[ [\alpha]_G = \{(u, v) \mid \exists \alpha \in \Sigma. (u, a, v) \in E\} \]

where \( \cdot \) is the symbol for the concatenation of binary relations and \( R^\epsilon \) denotes the concatenation of \( R \) with itself \( \epsilon \) times.

**Example III.1** Consider the graph of Example II.2 and the schema of Example II.7. The following query returns all pairs \((x, y)\) where \( x \) is the author of a paper in a conference series \( y \), but also published a paper in a journal \( z \):

\[ \text{[creator-} \cdot \text{journal} \cdot \text{creator-} \cdot \text{partOf} \cdot \text{series} \]

The result of this query is \( \{(\text{John E. Hopcroft, focus})\} \).

**Example III.2** Consider the graph depicted in Figure 2. This graph is typed by the following graph schema:

\[ e_1 = (a, a, kb?&c?&d?) \quad e_2 = (b, e) \quad e_3 = (c, e) \quad e_4 = (d, e) \]

Consider now the following query: \( \text{[creator-} \cdot \text{\{0,}\} \cap \epsilon \cdot \text{\{b, c\}} \)

This query selects all pairs of nodes \((x, y)\) where \( x \) is part of a cycle and \( y \) is reachable from \( x \) through an edge labelled with \( b \) or \( c \). The result of this query is \( \{(4, 2), (4, 6)\} \).

### IV. Inference Rules

In this section we sketch a type inference approach for typing GXPath queries. Indeed, a query is typed by a set of chains, where a chain is a sequence of \((s, e)\) pairs, each one containing a single path \( s \) and a schema element \( e \); each chain inferred for a query describes not only the path traversed in the graph by the query during its evaluation, but also the schema elements of the nodes reached by each step of the path. This information can be very helpful during the debugging process and for speeding up query evaluation.

**Example IV.1** Consider the query of Example III.1. The system infers the following chain set: \( \{(e_5, e_6), (creator, e_1), (partOf, e_3), (series, e_4)\} \). This chain set, containing a single chain, gives us very detailed
information about the nodes traversed during query evaluation. The first pair describes the starting point of the navigation, the second one says that a creator edge is traversed backward (from $c_5$ to $e_1$), while the third and the fourth ones describe forward navigation through edges partOf and series, by indicating the node types met at each step.

Chains are defined as follows.

**Definition IV.2** A chain $c$ over a graph schema $S$ is a finite sequence $\{(s_i, e_i)\}_{i=1,\ldots,n}$ such that:
- $s_1 = e$;
- $s_i = a \Rightarrow a \in \text{sym}(e_{i-1}.out) \cap \text{sym}(e_i.in)$;
- $s_i = a^- \Rightarrow a \in \text{sym}(e_{i-1}.in) \cap \text{sym}(e_i.out)$.

Static chain inference enables a precise vision of the behaviour of the query when evaluated over graphs of a schema $S$. At the same time, it poses some challenges. In the presence of recursive schemas and queries (like those in Example III.2), the inference could end up with an infinite set of chains, each one corresponding to a finite unfolding of the cyclic subgraph schema traversed by the query. Concerning Example III.2 we would have an infinite set of chains of the form:

- $c_1 = \{(e_1), (a, e_1), (b, e_2)\}$
- $c_2 = \{(e_1), (a, e_1), (c, e_3)\}$
- $c_3 = \{(e_1), (a, e_1), (a, e_1), (b, e_2)\}$
- $c_4 = \{(e_1), (a, e_1), (a, e_1), (a, e_1), (c, e_3)\}$

To avoid the derivation of infinite chain sets, we could use the following inference rule (where $C$ is a chain set):

\[
\frac{\vdash_{S} \alpha : C}{\vdash_{S} \alpha^{m \cdot \star} : \bigcup_{i=m}^{\infty} C^i}
\]

In this rule the Kleene star is unfolded $||S||$ times, where $||S||$ is the number of schema elements in $S$. This rule, hence, traverses each cycle only a finite number of times; the set of inferred chains describes a representative, finite subset of the possible infinite chains corresponding to a query. For the previous example, a finite inference could be $\{c_1, c_2\}$. This finite inference is useful for both debugging and optimisation purposes. It gives information about which types of nodes are traversed and in which order (debugging), and can guide query evaluation in order to avoid that useless nodes (e.g., those typed by $e_4$) are explored.

Another challenging query operator is complement. Consider again the graph of Example III.2 and the query $a \cdot b$. The result of this query is just $\{(3, 2)\}$ and can be typed by the chain set $\{((e_1), (a, e_1), (b, e_2))\}$. Consider now the query $a \cdot b^\ast$. The result of this query is the complement of the result of the previous query, hence it comprises any node pairs except for $(3, 2)$. In this case any sound inference rule for complement should compute chains traversing schema elements that are not related, e.g., $e_4$ and $e_3$; these chains should be not only sound, but also precise,\(^1\) which can be far from being obvious

\(^1\)Here: sound means that inferred chains include those that are actually traversed by the query in schema instances; precise means that chains that are never traversed in schema instances should not be inferred.

for loosely connected (or disconnected) schema elements.

Finally, dealing with intersection queries requires some care. To illustrate, consider queries $\alpha_1 = a \cdot a \cdot a \cdot a$ and $\alpha_2 = a \cdot a \cdot a \cdot a \cdot a \cdot a$ over graphs typed by the schema in Example III.2. When evaluated on the graph of Example III.2, the two queries result in the same pairs of nodes. Hence their intersection is not empty; however it is easy to verify that two disjoint sets of chains would be inferred from them. So, adopting set intersection to infer chains for the $\alpha_1 \cap \alpha_2$ query would be unsound. To solve this problem, we are currently considering a rule of the following form:

\[
\frac{\vdash_{S} \alpha_1 : C_1 \vdash_{S} \alpha_2 : C_2}{\vdash_{S} \alpha_1 \cap \alpha_2 : C_1 \cap C_2}
\]

where $\cap$ is defined as follows.

**Definition IV.3** Given two chain set $C$ and $C'$, we define $C \cap C'$ as:

\[
\{(e_1, \ldots, e_n) \mid \exists [(e_1, \ldots, e_n) \in C \land (e_1', \ldots, e_n') \in C'] \lor (e_1, \ldots, e_n) \in C \land (e_1', \ldots, e_n') \in C'\}
\]

In a nutshell, $C \cap C'$ retains a chain $c$ belonging to at least one of the two sets $C$ and $C'$ provided that there exists another chain $c'$ in the other set such that the first and the last schema elements of the two chains coincide. This ensures the following soundness property: if a pair of nodes is returned by an intersection query, then all possible chains the query has traversed to return this pair are inferred. We are currently investigating the interaction of such a rule with inference techniques for complement and counting.

**V. Conclusions and Future Work**

In this paper we briefly introduced a schema language for data graphs, and discussed the issues related to type inference for GXPath queries. In the very near future we want to formalize our characterization of empty schemas, to provide a correct and complete algorithm for detecting empty schemas, as well as to develop a sound and precise set of inference rules for GXPath queries.

**References**