Typing Query Languages for Data Graphs

Dario Colazzo #  Carlo Sartiani *

# LAMSADE, Université Paris-Dauphine
* DIMIE, Università della Basilicata

GDM 2014
Graph databases and query languages

- Graph databases: used in many application areas such as
  - Semantic Web
  - social networks
  - bioinformatics
  - network traffic analysis
  - crime detection
  - etc.

Several graph query languages have been proposed. We focus on GXPath [Libkin, Martens, Vrgoc - ICDT'14]:
- based on the popular XPath language
- extends existing languages (RPQ and NRE)
- allows for forward and backward navigation
- includes union, intersection, negation and forms of recursion
- has tractable combined complexity for query evaluation
Graph databases and query languages

- Graph databases: used in many application areas such as
  - Semantic Web
  - social networks
  - bioinformatics
  - network traffic analysis
  - crime detection
  - etc.

- Several graph query languages have been proposed
Graph databases and query languages

- Graph databases: used in many application areas such as
  - Semantic Web
  - social networks
  - bioinformatics
  - network traffic analysis
  - crime detection
  - etc.

- Several graph query languages have been proposed

- We focus on GXPath [Libkin, Martens, Vrgoc - ICDT’14 ]:
  - based on the popular XPath language
  - extends existing languages (RPQ and NRE)
  - allows for forward and backward navigation
  - includes union, intersection, negation and forms of recursion
  - has tractable combined complexity for query evaluation
Need for typing

- Existing graph query languages are essentially *untyped*.
Need for typing

- Existing graph query languages are essentially \textit{untyped}.
- Need for mechanisms to:
  - Infer the structure of query results (type inference)
  - Check whether query results meet a given schema (type checking)
  - Check whether query result is always empty (query correctness)
- Typing can aid:
  - Speed up development of correct queries
  - Query optimization
Need for typing

- Existing graph query languages are essentially *untyped*.
- Need for mechanisms to:
  - infer the structure of query results (type inference)
  - check whether query results meet a given schema (type checking)
  - check whether query result is always empty (query correctness)

Typing can aid:
- speed up development of correct queries
- query optimization
Need for typing

- Existing graph query languages are essentially *untyped*.
- Need for mechanisms to:
  - infer the structure of query results (type inference)
Need for typing

- Existing graph query languages are essentially *untyped*.
- Need for mechanisms to:
  - infer the structure of query results (type inference)
  - check whether query results meet a given schema (type checking)
Need for typing

- Existing graph query languages are essentially *untyped*.
- Need for mechanisms to:
  - infer the structure of query results (type inference)
  - check whether query results meet a given schema (type checking)
  - check whether query result is always empty (query correctness)
Need for typing

- Existing graph query languages are essentially *untyped*.
- Need for mechanisms to:
  - infer the structure of query results (type inference)
  - check whether query results meet a given schema (type checking)
  - check whether query result is always empty (query correctness)
- Typing can aid:
Need for typing

- Existing graph query languages are essentially *untyped*.
- Need for mechanisms to:
  - infer the structure of query results (type inference)
  - check whether query results meet a given schema (type checking)
  - check whether query result is always empty (query correctness)
- Typing can aid:
Need for typing

- Existing graph query languages are essentially *untyped*.
- Need for mechanisms to:
  - infer the structure of query results (type inference)
  - check whether query results meet a given schema (type checking)
  - check whether query result is always empty (query correctness)
- Typing can aid:
  - speed up development of correct queries
Need for typing

- Existing graph query languages are essentially *untyped*.
- Need for mechanisms to:
  - infer the structure of query results (type inference)
  - check whether query results meet a given schema (type checking)
  - check whether query result is always empty (query correctness)
- Typing can aid:
  - speed up development of correct queries
  - query optimization
Our approach

- Design and study an expressive notion of schemas for graphs.

Two graph nodes can be connected by multiple kinds of paths. It is important to statically infer structural information about paths that queries follow in order to compute their results. This paves the way to both optimisation and fast and effective query debugging possibilities.
Our approach

- Design and study an expressive notion of schemas for graphs.
- Design and study type inference algorithms for GXPath, able to:
  - Infer type information about both query behaviour and result.
  - Two graph nodes can be connected by multiple kinds of paths. It is important to statically infer structural information about paths that queries follow in order to compute their results.
  - This paves the way to both optimisation and fast and effective query debugging possibilities.
Our approach

- Design and study an expressive notion of schemas for graphs.
- Design and study type inference algorithms for GXPath, able to:
  - infer type information about both query behaviour and result
  - Two graph nodes can be connected by multiple kinds of paths
    it is important to statically infer structural information about paths that queries follow
    in order to compute their results
  - This paves the way to both optimisation and fast and effective query debugging possibilities.
Our approach

- Design and study an expressive notion of schemas for graphs.
- Design and study type inference algorithms for GXPath, able to:
  - Infer type information about both query behaviour and result
Our approach

- Design and study an expressive notion of schemas for graphs.
- Design and study type inference algorithms for GXPath, able to:
  - infer type information about both query behaviour and result
Our approach

- Design and study an expressive notion of schemas for graphs.
- Design and study type inference algorithms for GXPath, able to:
  - infer type information about both query behaviour and result
- Two graph nodes can be connected by multiple kinds of paths
Our approach

- Design and study an expressive notion of schemas for graphs.
- Design and study type inference algorithms for GXPath, able to:
  - infer type information about both query behaviour and result
- Two graph nodes can be connected by multiple kinds of paths
  - it is important to statically infer structural information about paths that queries follow in order to compute their results
Our approach

- Design and study an expressive notion of schemas for graphs.
- Design and study type inference algorithms for GXPath, able to:
  - infer type information about both query behaviour and result
- Two graph nodes can be connected by multiple kinds of paths
  - it is important to statically infer structural information about paths that queries follow in order to compute their results
- This paves the way to both optimisation and fast and effective query debugging possibilities.
Data Graphs

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.

Incoming and outgoing edges of a node:

- $in(v) = \{(v', a, v) \in E \mid v' \in V \land a \in \Sigma\}$;
- $out(v) = \{(v, a, v') \in E \mid v' \in V \land a \in \Sigma\}$. 
Data Graphs
Graph Schema $\mathcal{S} = \{e_i\}_{i=0}^n$

- $e_1 = (\epsilon, (\text{journal} + \text{partOf}) \& (\text{creator})^+)$
- $e_2 = (\text{journal}^*, \epsilon)$
- $e_3 = (\text{partOf}^*, \text{series})$
- $e_4 = (\text{series}^*, \epsilon)$
- $e_5 = (\text{creator}^*, \epsilon)$
Graph Schema $\mathcal{S} = \{e_i\}_{i=0}^n$

- Each schema element $e_i = (e.in, e.out)$, with $e.in$ and $e.out$ are regular expressions.
Graph Schema $\mathcal{S} = \{e_i\}_{i=0}^n$

- Each schema element $e_i = (e.in, e.out)$, with $e.in$ and $e.out$ are regular expressions.
- For the example $\mathcal{S} = \{e_1, e_2, e_3, e_4, e_5\}$;

\[
\begin{align*}
e_1 &= (\epsilon, (\text{journal} + \text{partOf}) \& (\text{creator})^+) \\
e_2 &= (\text{journal}^*, \epsilon) \\
e_3 &= (\text{partOf}^*, \text{series}) \\
e_4 &= (\text{series}^*, \epsilon) \\
e_5 &= (\text{creator}^*, \epsilon)
\end{align*}
\]
Schema Emptiness

We restrict to schemas $\mathcal{S} = \{e_i\}_{i=0}^n$ such that:

1. $\forall i \in [0..n]. \forall l \in \text{sym}(e_i.\text{in}). \exists j \in [0..n]. l \in \text{sym}(e_j.\text{out});$

2. $\forall i \in [0..n]. \forall l \in \text{sym}(e_i.\text{out}). \exists j \in [0..n]. l \in \text{sym}(e_j.\text{in});$

$\blacktriangleright$ $1 \land 2 \Rightarrow$ no valid graphs with dangling edges.
We restrict to schemas $\mathcal{S} = \{e_i\}_{i=0}^n$ such that:

1. $\forall i \in [0..n]. \forall l \in \text{sym}(e_i.\text{in}). \exists j \in [0..n]. l \in \text{sym}(e_j.\text{out})$;
2. $\forall i \in [0..n]. \forall l \in \text{sym}(e_i.\text{out}). \exists j \in [0..n]. l \in \text{sym}(e_j.\text{in})$;

$\implies 1 \land 2 \implies$ no valid graphs with dangling edges.
Schema Emptiness

We restrict to schemas $\mathcal{S} = \{e_i\}_{i=0}^n$ such that:

1. $\forall i \in [0..n]. \forall l \in \text{sym}(e_i.\text{in}). \exists j \in [0..n]. l \in \text{sym}(e_j.\text{out})$;
2. $\forall i \in [0..n]. \forall l \in \text{sym}(e_i.\text{out}). \exists j \in [0..n]. l \in \text{sym}(e_j.\text{in})$;

$\Rightarrow 1 \land 2 \Rightarrow$ no valid graphs with dangling edges.

$\Rightarrow$ Conditions 1 and 2 are not sufficient for non-emptiness.

Consider $\mathcal{S} = \{e_1, e_2\}$, with:

$$e_1 = (\epsilon, a\&b\&c\&c)$$
$$e_2 = (a\&b\&c, \epsilon)$$
We restrict to schemas $S = \{e_i\}_{i=0}^n$ such that:

1. $\forall i \in [0..n].\forall l \in \text{sym}(e_i.\text{in}).\exists j \in [0..n].l \in \text{sym}(e_j.\text{out})$;
2. $\forall i \in [0..n].\forall l \in \text{sym}(e_i.\text{out}).\exists j \in [0..n].l \in \text{sym}(e_j.\text{in})$;

- $1 \land 2 \Rightarrow$ no valid graphs with dangling edges.
- Conditions 1 and 2 are not sufficient for non-emptiness.
- Consider $S = \{e_1, e_2\}$, with:

$$
\begin{align*}
    e_1 & = (\epsilon, a\&b\&c\&c) \\
    e_2 & = (a\&b\&c, \epsilon)
\end{align*}
$$

- Checking emptiness is not obvious.
- How many $e_i$ nodes do we need to build a valid instance?
Schema Emptiness

Our approach: associate a system of linear equations to $S$, such that:

$S$ is not empty iff the system admits non-zero integer solutions.

System construction, main steps:

- Associate to each schema element $e_i$ a variable - the variable indicates how many nodes of that schema el. the instance contains
- Build, for each symbol $a$, a polynomial equation describing the produced and consumed edges labelled with $a$

Example, empty schema $S = \{ e_1, e_2 \}$,

\[
\begin{align*}
  e_1 &= (\epsilon, a\&b\&c\&c) \\
  e_2 &= (a\&b\&c, \epsilon)
\end{align*}
\]

\[
\begin{align*}
  a. & \quad x - y = 0 \\
  b. & \quad x - y = 0 \\
  c. & \quad 2x - y = 0
\end{align*}
\]
For the non-empty schema \( S = \{e_1, e_2, e_3\} \), with:

\[
\begin{align*}
    e_1 &= (\epsilon, a&b&c&c&c&c) \\
    e_2 &= (a&b&c, \epsilon) \\
    e_3 &= (c&c, \epsilon)
\end{align*}
\]

We have:

\[
\begin{align*}
    a. & \quad x - y = 0 \\
    b. & \quad x - y = 0 \\
    c. & \quad 4x - y - 2z = 0
\end{align*}
\]

- the non-zero, integer solution \((2,2,3)\) means that a valid graph with 2 \( e_1 \) vertices, 2 \( e_2 \) vertices, and 3 \( e_3 \) vertices exists.
GXPath

\[ \alpha ::= \epsilon \mid _ \mid a \mid a^\sim \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha^{m,n} \mid \alpha \cap \alpha \mid [\alpha] \mid \bar{\alpha} \]

Given \( G = (V, E, \rho) \), \([\alpha]_G\) consists of all pairs of nodes \((v, u)\) such that labeled-edges along a path connecting \( u \) to \( v \) in \( G \) can be traversed according to navigational specifications of \( \alpha \).
GXPath

\[
\alpha ::= \epsilon | _ | a | a^- | \alpha + \alpha | \alpha \cdot \alpha | \alpha^{m,n} | \alpha \cap \alpha | [\alpha] | \bar{\alpha}
\]

Given \( G = (V, E, \rho) \), \([\alpha]_G\) consists of all pairs of nodes \((v, u)\) such that labeled-edges along a path connecting \( u \) to \( v \) in \( G \) can be traversed according to navigational specifications of \( \alpha \).

\[
[\epsilon]_G = \{(u, u) \mid u \in V\}
\]
\[
[_]_G = \{(u, v) \mid \exists a \in \Sigma.(u, a, v) \in E\}
\]
\[
[a]_G = \{(u, v) \mid (u, a, v) \in E\}
\]
\[
[a^-]_G = \{(u, v) \mid (v, a, u) \in E\}
\]
\[
[\alpha_1 + \alpha_2]_G = [\alpha_1]_G \cup [\alpha_2]_G
\]
\[
[\alpha_1 \cdot \alpha_2]_G = [\alpha_1]_G \circ [\alpha_2]_G
\]
\[
[\alpha^{m,n}]_G = \bigcup_{i=m}^{n}[\alpha]^i_G
\]
\[
[\alpha_1 \cap \alpha_2]_G = [\alpha_1]_G \cap [\alpha_2]_G
\]
\[
[[\alpha]]_G = \{(u, u) \mid (u, v) \in [\alpha]_G\}
\]
\[
[\bar{\alpha}]_G = \{(u, v) \mid (u, v) \notin [\alpha]_G\}
\]
GXPath, example

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 \(z\):

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

Result: \(\{(John \ E. \ Hopcroft, \ focs)\}\).
Chain-based type inference

- Query: \([creator^− \cdot journal] \cdot creator^− \cdot partOf \cdot series\)
- Result: \(\{(John \ E. \ Hopcroft, focs)\}\)
- Inferred chain set:

\[
\{(\epsilon, e_5), (creator^−, e_1), (partOf, e_3), (series, e_4)\},
\{(\epsilon, e_5), (creator^−, e_1), (journal, e_2)\}\}
\]

\[
e_1 = (\epsilon, (journal + partOf) \& (creator)^+) \\
e_2 = (journal*, \epsilon) \\
e_3 = (partOf*, \text{series}) \\
e_4 = (series*, \epsilon) \\
e_5 = (creator*, \epsilon)
\]
Current and future work

- Complete the study of emptiness checking of graph-schemas
- Study type inference approaches for recursion and intersection:
  - $\alpha^m,*$
  - $\alpha_1 \cap \alpha_2$
- Extend the type inference technique in order to deal with GXPath negation:
  \[
  [\alpha]_G = \{(u, v) \mid (u, v) \notin [\alpha]_G\}
  \]
- Study main properties of the resulting type system.
  - soundness
  - precision
  - complexity of the typing algorithm
- Study the membership and schema inclusion problems.