GPC: A Pattern Calculus for Property Graphs

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3584372

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '23)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
ABSTRACT
The development of practical query languages for graph databases runs well ahead of the underlying theory. The ISO committee in charge of database query languages is currently developing a new standard called Graph Query Language (GQL) as well as an extension of the SQL Standard for querying property graphs represented by a relational schema, called SQL/PGQ. The main component of both is the pattern matching facility, which is shared by the two standards. In many aspects, it goes well beyond RPQs, CRPQs, and similar queries on which the research community has focused for years.

Our main contribution is to distill the lengthy standard specification into a simple Graph Pattern Calculus (GPC) that reflects all the key pattern matching features of GQL and SQL/PGQ, and at the same time lends itself to rigorous theoretical investigation. We describe the syntax and semantics of GPC, along with the typing rules that ensure its expressions are well-defined, and state some basic properties of the language. With this paper we provide the community a tool to embark on a study of query languages that will soon be widely adopted by industry.

CCS CONCEPTS
• Information systems → Graph-based database models; Query languages.

KEYWORDS
graph databases, graph query languages, GQL, SQL/PGQ, pattern matching, syntax and semantics, expressive power, complexity, type systems

1 INTRODUCTION
The foundations of graph databases were laid more than 30 years ago in papers that define the now ubiquitous notion of regular path queries (RPQs) [10, 12]. They preceded the development of graph database systems by decades: it was in this millennium that the graph database industry properly emerged, driven by two similar models, namely RDF data and property graphs. The latter one is now promoted by multiple vendors such as Oracle, Neo4j, Amazon, SAP, Redis, TigerGraph etc. A very notable development in the evolution of property graph databases is the decision, taken 3 years ago, to produce a new standard query language called GQL [24]. It would subsume the hitherto used languages such as Cypher [16] of Neo4j, PGQL [23] of Oracle, GSQL [14] of TigerGraph, and the G-CORE proposal from an industry/academia group [1].

The new GQL (Graph Query Language) Standard is developed by the same ISO committee that is in charge of developing and maintaining the SQL Standard. The core of any graph query language is its pattern matching engine, that finds patterns in graphs. GQL’s pattern matching facilities are in fact shared across two standards:
• SQL/PGQ, a new Part 16 of the SQL standard, that defines querying graphs specified as views over a relational schema;
• GQL, a standalone language for querying property graphs.

The development of GQL as a query language standard is rather different from SQL. The latter came out of a well-researched relational theory; relational calculus led to its declarative approach, while relational algebra formed the foundation of RDBMS implementations. But while GQL is “inspired” by the key developments of database research, they are not represented directly in the language, which itself is designed by an industry consortium. The GQL committee lists1 three main academic influences: regular path queries [12], Graph XPath [18], and regular queries on graphs [22] which are the regular closure of conjunctive RPQs. While these provided important initial orientation, the GQL development is

1https://www.gqlstandards.org/existing-languages
much more in line with industry-level languages such as Cypher, GSQL, and PGQL.

The theory of graph query languages on the other hand produced a multitude of languages based on RPQs: CRPQs, UCRPQs, 2UCRPQs, ECRPQs, just to name a few (see [2, 3] for many more). However, none of them can play the role of relational calculus with respect to the development of GQL, as they do not capture its key features with respect to both navigation and handling data.

Our goal then is to produce that missing piece, a theoretical language that underlies GQL and SQL/PGQ pattern matching and can be studied in the same way as the RPQ family has been studied over decades. Doing so has two significant difficulties:

1. GQL pattern matching is described by close to 100 pages of the Standard text which is not human-friendly; it is intended for developers implementing the language;
2. Even that text is not available to the research community: the relevant ISO standards will only be published in 2023 and even then will be behind a paywall.

The only publicly available source is [13]. It outlines the main features of GQL and SQL/PGQ pattern matching by means of several examples and thus hardly serves the purpose of introducing a pattern matching calculus to form the basis of further study of graph querying. Such a calculus should judiciously choose the key features leaving others for extensions. Think again about the SQL/relational calculus analogy. The latter does not have bag semantics, nulls, aggregates, full typing, and many other SQL features. We adopt a similar approach here. Our goal is to introduce a calculus that captures the essence of GQL pattern matching, without every single feature present. Similarly to relational calculus, we settle for the core set-semantics fragment without nulls or aggregates, each of which can be added as extensions.

What is new? We now outline the main differences between the currently existing theoretical languages and real-life pattern matching in property graphs that we need to capture.

- Much of the theoretical literature relies on an overly simplified model of graph databases as edge-labeled graphs; this is common in the study of RPQs and extensions [3]. Typical patterns in languages like Cypher combine graph navigation with querying data held in nodes and edges. With data values, the theoretical model of choice is data graphs [18]: in those, each node carries a single data value, similarly to data trees studied extensively in connection with tree-structured data [7]. Property graphs are yet more complex, as each edge or node can carry an arbitrary collection of key-value pairs.

- GQL patterns bind variables – in different ways – and use them to select patterns. As an example, consider a pattern (in our notation, to be introduced in Section 3) \((x : a) \xrightarrow{e} y : a\) that looks for \(b\)-labeled edges between two \(a\)-labeled elements. While producing a match, it binds variables \(x\) and \(y\) to the start and end-nodes of the edge, and \(e\) to the edge itself. If, on the other hand, we look at \((x : a) \xrightarrow{e} y : a\), then we match paths of length 1 or more (indicated by \(1.\infty\)) from \(x\) to \(y\). In this case \(e\) gets bound not to an end, but rather to a list of edges. This has immediate implications on conditions in which such a binding can be used.

- Property graphs are multigraphs (there can be multiple edges between two endpoint vertices), pseudographs (there can be an edge looping from a vertex to itself), and mixed, or partially directed graphs (as an edge can be directed or undirected). This means edges can be traversed in different directions, or traversals can be indicated as having no direction at all.

- To ensure the number of paths returned is finite, real-life languages put additional restrictions on paths, such as insisting that they be trails (no repeated edges, as in Cypher), simple (no repeated nodes), or shortest (as in G-Core [1]). Typically in research literature one considers each one of those semantics separately, but GQL permits mixing them.

- Similarly to Cypher, in GQL one can match paths and output them. In theoretical languages this feature is rather an exception [1, 5].

- Finally, one can apply conditions to filter matched paths. For example, after matching \((x : a) \xrightarrow{e} y : a\), one can apply a condition \(x.k = y.k\) stating that property \(k\) of both \(x\) and \(y\) is the same. Notice that we cannot talk similarly about properties of \(e\), as they are bound to a list.

Other features of patterns are those in regular languages: concatenation, disjunction, repetition; they can be applied on top of already existing patterns, similarly to [22].

Our main contribution is the Graph Pattern-Matching Calculus GPC that captures all the key features of GQL and SQL/PGQ pattern matching. Its syntax is described in Section 3 (after the definition of property graph concepts in Section 2). To ensure the well-definedness of its expressions, the calculus comes with a type system, given in Section 4. We give a formal semantics of GPC in Section 5, and provide a number of basic results on the complexity of the language, and its relationship with classical theoretical formalisms in Section 6. In Section 7 we outline some possible extensions and describe two concrete examples where theoretical studies of the language had a direct impact on the Standard as it was being written.

2 DATA MODEL

We take the standard (as currently adopted by the GQL Standard committee [17]) definition of property graphs. We assume disjoint countable sets \(\mathcal{N}, \mathcal{E}_d, \mathcal{E}_u\) of node, directed, and undirected edge ids, \(\mathcal{L}\) of labels, \(\mathcal{K}\) of keys, and \(\text{Const}\) of constants. A property graph is a tuple \(G = (\mathcal{N}, \mathcal{E}_d, \mathcal{E}_u, \lambda, \text{endpoints}, \text{src}, \text{tgt}, \delta)\) where

- \(\mathcal{N} \subset \mathcal{N}\) is a finite set of node ids used in \(G\);
- \(\mathcal{E}_d \subset \mathcal{E}_d\) is a finite set of directed edge ids used in \(G\);
- \(\mathcal{E}_u \subset \mathcal{E}_u\) is a finite set of undirected edge ids used in \(G\);
- \(\lambda : \mathcal{N} \cup \mathcal{E}_d \cup \mathcal{E}_u \rightarrow 2^{\mathcal{L}}\) is a labeling function that associates with every id a (possibly empty) finite set of labels from \(\mathcal{L}\);
- \(\text{src}, \text{tgt} : \mathcal{E}_d \rightarrow \mathcal{N}\) define source and target of a directed edge;
- \(\text{endpoints} : \mathcal{E}_u \rightarrow 2^{\mathcal{N}}\) so that \(\text{endpoints}(e)\) is 1 or 2 define endpoints of an undirected edge;
- \(\delta : (\mathcal{N} \cup \mathcal{E}_d \cup \mathcal{E}_u) \times \mathcal{K} \rightarrow \text{Const}\) is a partial function that associates a constant with an id and a key from \(\mathcal{K}\).

We use node and edge to refer to node ids and edge ids, respectively, and call a node \(u\) an \(t\)-node iff \(\ell \in \lambda(u)\); similarly for edges.
A path is an alternating sequence of nodes and edges that starts and ends with a node, that is, it is a sequence of the form

$$u_0e_1u_1e_2 \cdots e_nu_n,$$

where \(u_0, \ldots, u_n\) are nodes and \(e_1, \ldots, e_n\) are (directed or undirected) edges. Note that we allow \(n = 0\), in which case the path consists of a single vertex and no edges. For a path \(p\) we denote \(u_0\) as src \((p)\) and \(u_n\) as tgt \((p)\); we also refer to \(u_0\) and \(u_n\) as the path’s endpoints. The length of a path \(p\), denoted len \((p)\), is \(n\), i.e., the number of occurrences of edge ids in \(p\). We also use the term edgeless path to refer to a path of length zero. We spell paths explicitly as path\((u_0, e_1, \ldots, e_n, u_n)\). We denote the set of all paths by Paths.

A path in \(G\) is a path such that each edge in it connects the nodes before and after it in the sequence. More formally, it is a path \((u_0, e_1, u_1, e_2, \ldots, e_n, u_n)\) such that at least one of the following holds for each \(i \in [n]\):

(a) src \((e_i) = u_{i-1}\) and tgt \((e_i) = u_i\) in which case we speak of \(e_i\) as a forward edge in the path;
(b) src \((e_i) = u_i\) and tgt \((e_i) = u_{i-1}\) in which case we speak of \(e_i\) as a backward edge in the path;
(c) endpoints \((e_i) = \{u_{i-1}, u_i\}\) in which case we speak of \(e_i\) as an undirected edge in the path.

Here, both (a) and (b) can be true at the same time in the case of a directed self-loop. By Paths\((G)\) we denote the set of paths in \(G\). Notice that Paths\((G)\) can be infinite.

Two paths \(p = \text{path}(u_0, e_0, u_0, \ldots, u_k)\) and \(p' = \text{path}(u_0', e_0', \ldots, u_j)\) concatenate if \(u_k = u_0'\), in which case their concatenation \(p \cdot p'\) is defined as \(\text{path}(u_0, e_0, u_0, \ldots, u_k, e_0', \ldots, u_j)\). Note that if one of the paths consists of a single node, then it is a unit of concatenation and does not change the result. That is, \(p \cdot \text{path}(u)\) is defined if \(u = u_k\), in which case it equals \(p\); likewise for \(\text{path}(u) \cdot p\) and \(u = u_0\).

### 3 PATTERN CALCULUS

We assume a countably infinite set \(X\) of variables. The basic building blocks of GPC are node patterns and edge (or arrow) patterns. Node patterns are of the form \((x : \ell)\). Here \(x\) is a variable, and \(\ell\) specifies the node label. The brackets “(“ and “)” are mandatory, and signify that we are talking about a node. Both the variable \(x\), and the label specification \(x : \ell\) are optional, and can be omitted. This way, the simplest node pattern is \(()\), matching any node in the graph. The presence of a variable means that it gets bound; the presence of a label \(\ell\) means that only \(\ell\)-nodes are matched. Edge patterns are of the form \(\equiv\), where again \(x\) and \(\ell\) are a variable and an edge label, respectively, and \(\equiv\) is one of the allowed directions: \(\rightarrow\) (forward), \(\leftarrow\) (backward), \(\equiv\) (undirected). Both \(x\) and \(\ell\) can be omitted. In the case when they are present, the variable \(x\) gets bound to the matching edge, and \(\ell\) constrains the allowed edge labels.

The concrete syntax of GQL and SQL/PGQ involves aspects like parentheses and precedence of operators, which we do not take into account here. Instead, we assume that GPC expressions are already parsed, so as to work directly on their abstract syntax trees. The full grammar of GPC is given in Fig. 1. Here:

---

2As is usual in the graph database literature \([3, 20, 21, 25]\), we use the term path to denote what is called walk in the graph theory literature \([8]\).

---
Examples. The formal semantics is presented in Section 5. Next we illustrate how GPC operates with several examples. Each path pattern is matched to a path; such a path could be a single node, an edge (with endpoints included), or a more complex path. For example, consider the pattern

\[(x_1 : A) \xrightarrow{\Pi} (x_2 : B) \xrightarrow{\Pi} (x_3 : C) \xrightarrow{\Pi} (x_1)\]

matches a path from an A-node to itself via B- and C-nodes with the first and third edges going forward and the second edge going backward. Notice that this pattern introduces an implicit join over the endpoints of the path by repeating the variable \(x_1\).

The pattern \((x : A) \xrightarrow{\Pi} (z : B) [\iff (u : C) + ()]\) is an optional pattern\(^1\): a feature present in many languages such as SPARQL and Cypher. It matches an edge from an A-node to a B-node, binding \(x\) and \(z\) to its endpoints, and if the B-node has an incoming edge from a C-node, binds \(u\) to its source. This pattern can be seen as the disjunction \(\pi_1 + \pi_2\) where \(\pi_1 = (x : A) \xrightarrow{\Pi} (z : B) [\iff (u : C)]\) and \(\pi_2 = (x : A) \xrightarrow{\Pi} (z : B) [\iff ()]\). In \(\pi_2\), the concatenated \(\iff\) must match the same node as \(z\) and thus it has no effect on the pattern; this accounts for the case where the node bound to \(z\) has no incoming edge from a C-node.

The pattern \((x : A) \xrightarrow{\Pi} (z : B)\) looks for paths of arbitrary positive length from an A-node to a B-node. It uses variable \(y\) to bind edges encountered on this path. Unlike the bindings for \(x\) and \(z\), which are unique nodes, there may well be multiple edges encountered on the path between them, and thus \(y\) needs to be bound to a complex object encoding all such edges on a path. Intuitively, in this case \(y\) binds to the list of edges on a matching path. However, in general the binding for such variables, which we call group variables, is more complex. Indeed, consider a pattern \(\pi^n \cdot m\) and a particular match of this pattern in which \(\pi\) is repeated \(k\) times, \(n \leq k \leq m\). These \(k\) repetitions of \(\pi\) are matched by paths \(p_1, \ldots, p_k\), and thus for every variable used in \(\pi\) we need to record not only which elements of \(p_1, \ldots, p_k\) it binds to, but also in which paths \(p_i\) these elements occur. Thus, in general, group variables will be bound to lists of (path, graph element) pairs.

The pattern \([(x : A) \xrightarrow{\Pi} (z : B)](x.a = z.a)\) is an example of a conditioned pattern; here the condition ensures that the value of property \(a\) is the same at the endpoints of the path. Note that conditions cannot compare nodes or edges, only their properties.

A pattern cannot be used by itself as a query; for example if we write \(u = [(x : A) \xrightarrow{\Pi} (z : B)]\) then the variable \(u\) can be bound to infinitely many paths. Indeed, if there is a loop on some path from \(x\) to \(z\), it can be traversed arbitrarily many times, while still satisfying the condition of the pattern. To deal with this, every pattern in a query is compulsorily preceded by a restrictor, e.g., \(u = \text{trail}[(x : A) \xrightarrow{\Pi} (z : B)]\). Then only trails, of which there are finitely many, that satisfy the conditions of \(\pi\) will be returned as values of variable \(u\).

The necessity of type rules. The calculus defined in Fig. 1 is very permissive and allows expressions that do not type-check. For example, \((x) \xrightarrow{\Pi} ()\) is syntactically permitted even though it equates a node variable with an edge variable. As another example, adding the condition \(x.a = y.a\) to the pattern \((x : A) \xrightarrow{\Pi} (z : B)\) seen above would result in comparing a singleton with a list of pairs. The type system introduced next eliminates such mismatches.

4 TYPE SYSTEM

The goal of the type system is to ensure that GPC expressions do not exhibit the pathological behavior explained at the end of the previous section.

The set \(T\) of types used to type variables is defined by the following grammar

\[\tau ::= \text{Node} | \text{Edge} | \text{Path} | \text{Maybe}(\tau) | \text{Group}(\tau).\]

The three atomic types are used for variables returning nodes, edges, and paths, respectively. The type constructor \(\text{Maybe}\) is used for variables occurring on one side of a disjunction only, while \(\text{Group}\) is used for variables occurring under repetition, whose bindings are grouped together. As variables in GPC are never bound to property values, we do not need the usual types like integers or strings. However, to eliminate references to unbound variables, we do need to type conditions (such as for \((x.a = y.a)\) in the example at the end of Section 3); we use an additional type \(\text{Bool}\) for that.

Typing statements are of the form \(\xi \vdash x : \tau\) stating that in expression \(\xi\) (a pattern or a query), we can derive that variable \(x\) has type \(\tau\), and \(\xi \vdash \theta : \text{Bool}\), stating that a condition is correctly typed as a Boolean value under the typing of other variables.

The typing rules are presented in Figure 2. Here, \(\text{var}(\xi)\) stands for the set of variables used in expression \(\xi\). For a type \(\tau\) we let \(\tau? = \tau\) if \(\tau = \text{Maybe}(\tau')\) for some \(\tau'\) and \(\tau? = \text{Maybe}(\tau)\) otherwise.

The five first rules state that variables in node/edge patterns, and variables naming paths, are typed accordingly. The next four rules say that variables of group type appear in repetition patterns, and that restrictors and path naming do not affect typing.

The next two lines deal with typing conditions: property values of singletons can be compared for equality; conditions are closed under Boolean connectives; and correctly typed conditions do not affect the typing of variables in a pattern.

The following two lines deal with the optional type \(\text{Maybe}(\tau)\). It is assigned to a variable \(z\) in a disjunction \(\pi_1 + \pi_2\) if in one of the patterns \(z\) is of type \(\tau\) and in the other \(z\) is either not present or of type \(\text{Maybe}(\tau)\).

Derivation rules for concatenation \(\pi_1 \pi_2\) and join \(Q_1, Q_2\) are similar: a variable is allowed to appear in both expressions only if it is typed as a node or an edge in both, or it inherits its type from one when it does not appear in the other.

Definition 1. An expression is well-typed if for every variable used in it, its type can be derived according to the typing rules. A well-typed expression assigns a unique type to every variable appearing in it, and only to such variables.

Proposition 2. For every well-typed expression \(\xi\), variable \(x\), and types \(\tau, \tau'\),

1 Note that we use square brackets for grouping, since \(\iff\) defines a node pattern.
### Figure 2: Typing rules for the GPC type system.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type System</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x) \vdash x : \text{Node})</td>
<td>(\pi \vdash z : \tau)</td>
<td></td>
</tr>
<tr>
<td>((x : \ell) \vdash x : \text{Node})</td>
<td>(\pi^n \vdash z : \tau)</td>
<td></td>
</tr>
<tr>
<td>((x) \iff \vdash x : \text{Edge})</td>
<td>(\rho \vdash z : \tau)</td>
<td></td>
</tr>
<tr>
<td>(\ell \iff \vdash x : \text{Edge})</td>
<td>(\rho \vdash z : \tau)</td>
<td></td>
</tr>
</tbody>
</table>

\(\pi \vdash \emptyset : \text{Bool}\)

\(\pi \vdash \theta \land \theta' : \text{Bool}\)

\(\pi \vdash \theta \lor \theta' : \text{Bool}\)

\(\pi \vdash \theta : \text{Bool}\)

\(\pi \vdash \neg \theta : \text{Bool}\)

\(\pi \vdash x.a = c : \text{Bool}\)

\(\pi \vdash x.a = y.b : \text{Bool}\)

### Definition 3.
Let \(\Theta\) denote a binary operator from GPC (Fig. 1). We say that \(\Theta\) is associative (resp. commutative) with respect to the type system if the condition (1) (resp. (2)) below holds for all expressions \(\xi_1, \xi_2, \xi_3, \Theta\), and variables \(x\):

\[\xi_1 \Theta \xi_2 \Theta \xi_3 \iff \xi_1 \Theta \xi_2 \Theta \xi_3 + x : \tau,\]

\[\xi_1 \Theta \xi_2 \Theta \xi_3 \iff \xi_2 \Theta \xi_1 \Theta \xi_3 + x : \tau.\]

### Proposition 4.

- **Union, concatenation and join are associative and commutative with respect to the type system.**
- **There is no expression \(\xi\) variable \(x\), and type \(\tau\) such that \(\xi : x : \tau \iff \text{Maybe}(\text{Maybe}(\tau))\).**

### Definition 5.
Given a well-typed expression \(\xi\), the *schema of \(\xi\)*, written \(\text{sch}(\xi)\), is the schema that maps each variable \(x \in \text{var}(\xi)\) to the unique type \(\tau\) such that \(\xi : x : \tau\). A variable \(x \in \text{var}(\xi)\) is called:

- a *singleton* variable if \(\text{sch}(\xi)(x) \in \{\text{Node}, \text{Edge}\}\);
- a *conditional* variable if \(\text{sch}(\xi)(x) = \text{Maybe}(\tau)\) for some \(\tau\);
- a *group variable* if \(\text{sch}(\xi)(x) = \text{Group}(\tau)\) for some \(\tau\);
- a *path variable* if \(\text{sch}(\xi)(x) = \text{Path}\).

It is easily checked that the function \(\text{sch}\) is *compositional*, in the following sense.

### Proposition 6.
For each binary operator \(\Theta\) of GPC, there exists a function that depends only on \(\Theta\), takes as arguments \(\text{sch}(\xi_1)\) and \(\text{sch}(\xi_2)\), and computes \(\text{sch}(\xi_1 \Theta \xi_2)\). Likewise for unary operators.

### 5 SEMANTICS

We begin by defining *values*, which is *what can be returned by a query*. Since GPC returns references to graph elements, not the data they bear, elements of \(\text{Const}\) are not values.

### Definition 7.
Given a type \(\tau \in \mathcal{T}\), the set \(\mathcal{V}\) of values of type \(\tau\) is defined inductively as follows:

- \(\mathcal{V}_{\text{Node}} = \mathcal{N}\), \(\mathcal{V}_{\text{Edge}} = \mathcal{E}_{\mathcal{V}}\), \(\mathcal{V}_{\text{Path}}\) = Paths;
- \(\mathcal{V}_{\text{Maybe}(\tau)} = \mathcal{V}_{\tau} \cup \{\text{Nothing}\}\) for a special value Nothing;
- \(\mathcal{V}_{\text{Group}(\tau)}\) is the set of all composite values of the form

\[
\text{list}\left(\{(p_0, v_0), \ldots, (p_n, v_n)\}\right)
\]

where \(n \geq 0\) and \(p_i \in \mathcal{V}_{\text{Path}}\) and \(v_i \in \mathcal{V}\) for all \(i \in [1, n]\).

The set of all values is \(\mathcal{V} = \bigcup_{\tau \in \mathcal{T}} \mathcal{V}_{\tau}\).

The semantics of GPC is defined in terms of *assignments* binding variables to values. Formally, an assignment \(\mu\) is a partial function from \(\mathcal{X}\) to \(\mathcal{V}\), with finite domain. We write \(\mu\) for the empty assignment; that is, an assignment that binds no variables. The values bound to variables of a well-typed pattern or query should respect its schema: we say that an assignment \(\mu\) conforms to a schema \(\sigma\) if \(\text{dom}(\mu) = \text{dom}(\sigma)\) and \(\mu(x) \in \mathcal{V}_{\text{sch}(\sigma)}\) for all \(x \in \text{dom}(\mu)\).

Given two assignments \(\mu\) and \(\mu'\), we say that \(\mu\) and \(\mu'\) unify if \(\mu(x) = \mu'(x)\) for all \(x \in \text{dom}(\mu) \cap \text{dom}(\mu')\). In that case, we define their unification \(\mu \cup \mu'\) by setting \(\mu \cup \mu'(x) = \mu(x)\) if \(x \in \text{dom}(\mu)\) and \(\mu(x) = \mu'(x)\) otherwise, for all \(x \in \text{dom}(\mu) \cup \text{dom}(\mu')\). If \(S\) is a family of assignments that pairwise unify, then their unification is associative, and we write it as \(\bigcup_{\mu \in S} \mu\).
The typing system ensures that all variables shared by $G$ are of the same type. For the sake of brevity, here we write $\mu$ and $\nu$ for $\rho$ and $\pi$ in Proposition 6, sch($\xi$) can be computed compositionally independently of $[\xi]_G$. In what follows, we shall define $[\xi]_G$ using $\text{sch}(\xi)$ and $[\xi']_G$ for direct subexpressions $\xi'$ of $\xi$. Hence, the semantics of expressions (patterns and queries), i.e., the function $[\xi]_G = (\text{sch}(\xi), [\xi']_G)$, is compositional.

For the remainder of this section, we consider a fixed property graph $G = (N, E_d, E_u, \lambda, \text{endpoints}, \text{src}, \text{tgt}, \delta)$.

**Semantics of atomic patterns.** For the sake of brevity, here we write atomic patterns as if all components were present, but still allow the possibility that some of them may be absent. Hence, $(x : \ell)$ subsumes the cases $(x) : \ell$ and $(\ell : x)$.

$$[x : \ell]_G = \{ (\text{path}(n), \mu) \mid n \in \ell, \ell \in \lambda(n) \text{ if } \ell \text{ is present} \}$$

where $\mu = \{ x \mapsto n \}$ if $x$ is present, and $\mu = \emptyset$ otherwise.

$$[x \rightarrow \ell]_G = \{ \text{path}(u_1, e, u_2), \mu' \mid e \in E_d, u_1 = \text{src}(e), u_2 = \text{tgt}(e), e \in \lambda(e) \text{ if } \ell \text{ is present} \}$$

$$[x \leftarrow \ell]_G = \{ \text{path}(u_2, e, u_1), \mu' \mid e \in E_d, u_1 = \text{src}(e), u_2 = \text{tgt}(e), e \in \lambda(e) \text{ if } \ell \text{ is present} \}$$

$$[x \leftrightarrow \ell]_G = \{ \text{path}(u_1, e, u_2), \mu' \mid e \in E_u, \text{endpoints}(e) = \{ u_1, u_2 \}, e \in \lambda(e) \text{ if } \ell \text{ is present} \}$$

where $\mu' = \{ x \mapsto e \}$ if $x$ is present, and $\mu' = \emptyset$ otherwise. Observe that the pattern $x \rightarrow \ell$ returns both path($u_1, e, u_2$) and path($u_2, e, u_1$) if endpoints($e$) = {$u_1, u_2$} with $u_1 \neq u_2$, but only one of them if $u_1 = u_2$ since both paths are the same.

**Semantics of concatenation.**

$$[p_1 \cdot p_2]_G = \{ (p_1, p_2, \mu_1 \cup \mu_2) \mid (p_1, \mu_1) \in [p_1]_G \text{ for } i = 1, 2, p_1 \text{ and } p_2 \text{ unify} \}$$

The typing system ensures that all variables shared by $p_1$ and $p_2$ are singleton variables (otherwise $p_1 \cdot p_2$ would not be well-typed). In other words, implicit joins over group and optional variables are disallowed (path variables do not occur in patterns at all).

**Semantics of union.**

$$[p_1 + p_2]_G = \{ (p, \mu \cup \mu') \mid (p, \mu) \in [p_1]_G \cup [p_2]_G \}$$

where $\mu'$ maps every variable in $\text{dom} \text{(sch}(p_1 + p_2)) \setminus \text{dom}(\mu)$ to Nothing. Note that here we rely on $\text{sch}(p_1 + p_2)$.

**Semantics of conditioned patterns.**

$$[\xi^\theta]_G = \{ (p, \mu) \in [\xi]_G \mid \mu \models \theta \}$$

where $\mu \models \theta$ is defined inductively as follows:

- $\mu \models (x.a = c)$ iff $\delta(\mu(x), a)$ is defined and equal to $c$;
- $\mu \models (x.a = y.b)$ iff $\delta(\mu(x), a)$ and $\delta(\mu(y), b)$ are defined and equal;
- $\mu \models (\theta_1 \land \theta_2)$ iff $\mu \models \theta_1$ and $\mu \models \theta_2$;
- $\mu \models (\theta_1 \lor \theta_2)$ iff $\mu \models \theta_1$ or $\mu \models \theta_2$;
- $\mu \models (\neg \theta)$ iff $\mu \not\models \theta$.

**Semantics of repeated patterns.**

$$[\pi_1 \cdot \pi_2]_G = \bigcup_{i=n}^m [\pi_i]_G$$

Above, for a pattern $\pi$ and a natural number $n \geq 0$, we use $[\pi]^n_G$ to denote the $n$-th power of $[\pi]_G$, defined as follows. We let

$$[\pi]^n_G = \{ \text{path}(n), \mu \mid u \text{ is a node in } G \}$$

where $\mu$ is the assignment that maps each variable in $\text{dom}(\text{sch}(\pi))$ to list(), the empty composite value. For $n > 0$, we let

$$[\pi]^n_G = \left\{ (p, \mu) \mid \begin{array}{l} (p_1, \mu_1), \ldots, (p_n, \mu_n) \in [\pi]^n_G \end{array} \right\}$$

where collect(($p_1, \mu_1), \ldots, (p_n, \mu_n)$) is an assignment defined, and discussed, below. In the case that $\pi$ does not have any variables, collect simply returns a function with empty domain. If $\pi$ does contain variables, then each such variable is mapped to a list. (As such, nesting of patterns of the form $\pi^{n-1}$ leads to nesting of lists.)

There are several ways to define collect and obtain a sound semantics. In all cases, collect takes as input any number of path/bind-
ing pairs ($p_1, \mu_1), \ldots, (p_n, \mu_n)$, such that $n > 0$ and $p_1, \ldots, p_n$ concatenate to a path $p = p_1 \cdots p_n$. Furthermore, by our inductive defi-
nition of the semantics, it will always be the case that $\mu_1, \ldots, \mu_n$ all have the same domain, that we denote by $D$. Then, collect(($p_1, \mu_1), \ldots, (p_n, \mu_n)$) is an assignment that maps every $x \in D$ to a list($p_1, \mu_1, \ldots, (p_n, \mu_n)$) where each $p_i$ is a portion of the matched path (they collectively satisfy $p_1, \ldots, p_n = p_1 \cdots p_n$) and $v_i$ is the value associated to $x$ for that portion.

If all $p_i$'s have a positive length (i.e., have at least one edge), collect is simply defined as follows.

$$\forall x \in D \ text{collect}((p_1, \mu_1), \ldots, (p_n, \mu_n))(x) = \text{list}((p_1, \mu_1(x)), \ldots, (p_n, \mu_n(x)))$$

(3)

Although collect is still well defined by (3) if some of the $p_i$'s have length 0, the above definition may lead to infinite query results. To avoid this, we outline three different approaches.

**Approach 1: Syntactic restrictions.** We add a syntactic restriction that prevents the case from ever appearing: pattern $\pi^{n-1}$ is forbid-
den if pattern $\pi$ may match an edgeless path. The latter is defined inductively: every edge pattern is allowed; if $\pi$ is allowed then so are $\pi(\theta), \pi^{n-1}, \pi' \pi$ and $\pi^\theta$ for every condition $\theta, n > 0$, and pattern $\pi'$: if $\pi_1$ and $p_2$ are allowed then so is $\pi_1 + p_2$. This is the solution adopted by the Cypher standard: the minimum path length of $\pi$ must be positive. The drawback of this solution is that it rules out syntactically some patterns for which (3) would result in a well-defined finite semantics.
\[ p = p_1 \circ p_2 \circ p_6 \circ p_6 \]

\[ p = p'_1 \circ p'_2 \circ p'_4 \circ p'_6 \]

**Figure 3:** Refactorization of a path \( p = p_1p_2 \cdots p_6 \) as \( p = p'_1p'_2 \cdots p'_7 \) by grouping consecutive edgeless factors.

**Approach 2: Run-time restriction.** As an alternative, the preconditions for `collect` well-definedness can be checked at run-time, i.e., it is only defined if all \( p_i \)'s have a positive length. While not imposing any additional restrictions, this approach has a drawback that it may have some result while \( \pi_{1} \cdot \pi \) has none, for some pattern \( \pi \).

**Approach 3: Grouping edgeless paths.** To overcome problems with the first two approaches, we propose a more general semantics of `collect` that groups together consecutive edgeless paths from \( p_1, \ldots, p_n \). If no such paths exist, either due to syntactic restriction or ruling them out at run-time, the result of this approach coincides with (3); thus this approach subsumes the other two.

We define \( p'_1, \ldots, p'_t \) as a coarser factorization of \( p_1, \ldots, p_n \): each \( p'_i \) is the concatenation of successive \( (p_j) \)'s, in which consecutive edgeless paths are grouped together, as shown in Figure 3. Formally, the \( p'_i \)'s are defined as the unique path sequence such that there exists \( i_1 < i_2 < \cdots < i_{k+1} \) (delineating the boundaries of \( p'_1, \ldots, p'_t \)) with \( i_1 = 1, i_{k+1} = n + 1 \) and satisfying the following:

\[
\begin{align*}
\forall k & \in \{1, \ldots, t\} \quad p'_k = p_{i_k} \cdots p_{i_{k+1} - 1} \\
\forall k & \in \{1, \ldots, t - 1\} \quad \text{len}(p_{i_k}) \neq 0 \lor \text{len}(p_{i_{k+1}}) \neq 0 \\
\forall k & \in \{1, \ldots, t\} \quad \text{either} \quad i_{k+1} = i_k + 1 \quad \text{and} \quad \text{len}(p_{i_k}) \neq 0 \\
& \quad \text{or} \quad \forall i, i_k \leq i < i_{k+1}, \text{len}(p_i) = 0
\end{align*}
\]

The assignment `collect`\((p_1, \mu_1), \ldots, (p_n, \mu_n)\) is defined only if

\[
\forall k \in \{1, \ldots, t\} \quad \mu_{i_k}, \ldots, \mu_{i_{k+1} - 1} \text{ pairwise unify}
\]

Then their unification is denoted by \( \delta'_k \) and `collect` is defined by

\[
\forall x \in D \quad \text{collect}((p_1, \mu_1), \ldots, (p_n, \mu_n))(x) = \text{list}((\delta'_1, \mu'_1(x)), \ldots, (\delta'_t, \mu'_t(x)))
\]

**Remark 8.** For the purpose of `collect`, one could use a weaker definition for unification that would allow \( \mu \) and \( \mu' \) to unify if, for every \( x \in \text{dom}(\mu) \cap \text{dom}(\mu') \), any of the following holds: \( \mu(x) = \text{Nothing} \), \( \mu'(x) = \text{Nothing} \) or \( \mu(x) \neq \mu'(x) \). This would allow even more combinations at the definition above.

Semantics of queries:

\[
\| \text{trail} \pi \|_G = \begin{cases} 
(p, \mu) \in \| \pi \|_G & \text{no edge occurs more than once in } p \\
\end{cases}
\]

\[
\| \text{simple} \pi \|_G = \begin{cases} 
(p, \mu) \in \| \pi \|_G & \text{no node occurs more than once in } p \\
\end{cases}
\]

\[
\| \text{shortest} \xi \|_G = \begin{cases} 
(p, \mu) \in \| \xi \|_G & \text{len}(p) = \min \{ \text{len}(p'), \text{src}(p') = \text{src}(p), \text{tg}(p') = \text{tg}(p) \} \\
\end{cases}
\]

where \( \xi \) is a trail, a path or simple \( \pi \), for some pattern \( \pi \). We then define:

\[
\| x = \rho \pi \|_G = \begin{cases} 
(p, \mu) \cup \{ (x, \pi) \} & (p, \mu) \in \| \rho \pi \|_G \\
\end{cases}
\]

\[
\| Q_1, Q_2 \|_G = \begin{cases} 
(p_1 \times p_2, \mu_1 \cup \mu_2) & (p_1, \mu_1) \in \| Q_i \|_G \text{ for } i = 1, 2 \\
\end{cases}
\]

Here, \( \rho_1 = (p_1^1, p_1^2, \ldots, p_1^k) \) and \( \rho_2 = (p_2^1, p_2^2, \ldots, p_2^k) \) are tuples of paths, and \( p_1 \times p_2 \) stands for \( (p_1^1, p_2^1), (p_1^2, p_2^2), \ldots, (p_1^k, p_2^k) \). Note that \( p_1 \) is a single path when \( Q_1 \) does not contain the join operator. Moreover, like for concatenation, the typing system guarantees that \( Q_1 \) and \( Q_2 \) are only joined over singleton variables, but not over path, group, or conditional variables.

In the results below we assume the third approach to the definition of `collect` as subsuming the other two. One may verify, by routine inspection, that the semantics is consistent with the typing system.

**Proposition 9.** For every well-typed expression \( \xi \) and every \( (\rho, \mu) \) in \( \| \xi \|_G \), all paths in \( \rho \) belong to Paths(\( G \)) and \( \mu \) conforms to sch(\( \xi \)).

Even though the set Paths(\( G \)) may be infinite, syntactic restrictions ensure finiteness of output.

**Theorem 10.** \( \| Q \|_G \) is finite for every query \( Q \) and graph \( G \).

**Proof Sketch.** We will treat only the case when \( Q = \rho \pi \); other cases follow from this case or are straightforward.

Let us first show that the set \( P = \{ p \mid 3\mu (p, \mu) \in \| Q \|_G \} \) is finite. If \( \rho \) is one of trail, simple, shortest trail or shortest simple, the claim holds since there are finitely many trails and simple paths in a graph. The last case, that is \( \rho = \text{shortest} \), follows from the fact that for all nodes \( s \) and \( t \) the set

\[
P_{(s, t)} = \{ p \in P \mid p \text{ starts in } s \text{ and ends in } t \}
\]

is finite. Indeed, all paths in \( P_{(s, t)} \) have the same length, and there are finitely many paths of a given length in a graph.

The remainder of the proof of Theorem 10 amounts to showing that for each path \( p \in P \), there are finitely many \( \mu \) such that \( (p, \mu) \in \| Q \|_G \). The proof of that claim is done by induction, the only nontrivial case is for patterns of the shape \( \pi_n \cdot \pi_m^\om 

**6 EXPRESSIVITY AND COMPLEXITY**

Expressive power. First, we look at the expressive power of GPC. For this, we will compare GPC with the main graph query languages considered in the research literature. Specifically, we compare GPC with regular path queries (RPQs) [12, 20], and their two-way expansion, 2RPQs [9]. In essence, an RPQ is specified via a regular expression and returns all pairs of nodes connected by a path whose edge labels form a word in the language of this expression. 2RPQs also allow traversing edges in the reverse direction, similarly to 2-APA in GPC, for a label \( a \). Two natural extensions are C2RPQs, which close 2RPQs under conjunctions [9, 10], and their unions, called UC2RPQs [10]. An interesting class is also that of nested regular expressions (NREs) [6], where along a path conforming to a regular language, we can test if there is an outgoing path conforming to another regular expression, as in PDL or XPath. Finally, we consider the class of regular queries (RQs) [22], which subsumes all the aforementioned classes. A regular query is a non-recursive Datalog program that is allowed to use transitive atoms of the form \( R^+(x, y) \)
in the body of the rules, where \( R \) is a binary predicate, either built in, or defined in the program.

In order to compare with the aforementioned languages, we consider a simple extension of GPC with projection and union, reflecting the fact that the pattern matching mechanism we formalize will be a sublanguage of a fully-fledged query language like GQL or SQL/PQG. A GPC+ query is a set of rules

\[
\text{Ans}(\vec{x}) := Q_1; \quad \text{Ans}(\vec{x}) := Q_2; \quad \ldots \quad \text{Ans}(\vec{x}) := Q_k
\]

where \( Q_i \) is a GPC query such that \( \vec{x} \subseteq \text{var}(Q_i) \) for all \( i \). The semantics of such a query on graph \( G \) is

\[
\left[ Q_1 \right]_G^x \cup \left[ Q_2 \right]_G^x \cup \cdots \cup \left[ Q_k \right]_G^x
\]

where \( \left[ Q_i \right]_G^x = \{ \mu(\vec{x}) \mid \exists p, \mu \in \left[ Q_i \right]_G \} \) for all \( i \). Notice that in our definition we allow unions only at the top level in order to combine results of queries whose arity is higher than binary. Binary unions are already covered at the level of GPC patterns, and can be arbitrarily nested inside iterations.

**Theorem 11.** GPC+ can express all of the following:
- unions of conjunctive two-way regular path queries (UC2RPQs);
- nested regular expressions (NREs);
- regular queries.

**Proof sketch.** For 2RPQs, note that these are explicitly present in the syntax of GPC patterns, and projecting on the endpoints gives us an equivalent expression. UC2RPQs and their unions are then handled by the conjunction of GPC queries, and unions in GPC+, respectively. The case of NREs is a bit more interesting, and it contains the blueprint for regular queries. To illustrate the main ideas, consider the nested regular expression \( (a[b^+][c]^*)^\omega \), which looks for paths of the form \( (ac)^\omega \), where after traversing an \( a \), we also check the existence of a nonempty path labelled with \( bs \). An equivalent GPC+ query is \( \text{Ans}(x,y) := Q \) where \( Q \) is the GPC query

\[
\text{shortest}(x) \xrightarrow{a^+} (z) \xrightarrow{b^-} 1.\infty (x) \leftrightarrow 1.\infty (z) \xrightarrow{x^-} 1.\infty (y).
\]

Basically, we introduce a fresh variable \( z \) which binds to the node from which we need to find a nonempty \( b \)-labelled path to an anonymous node, and then we return to the same node, thus allowing us to encode an answer inside of a single path. Since we only care about the endpoints in all of these query classes, the restrictor shortest is enough. This idea is then applied inductively in order to capture regular queries.

**Complexity.** When it comes to evaluating graph queries, one is used to dealing with high complexity. For instance, checking whether there is a query answer with a restrictor on top is known to be NP-hard (in data complexity) [3, 4, 19, 20, 25], and yet this feature is supported both by the GQL standard [13] and by concrete languages such as Cypher [16]. Accepting such high complexity bounds probably stems from the fact that query answers can be large in the case of graph queries. E.g., the number of shortest paths between two given nodes in a graph with \( O(n) \) nodes can be \( 2^n \). Given that query answers can be exponentially large (and in some cases, albeit not in GPC, even infinite), living with high complexity bounds is simply a fact of life.

In this light, we provide some insights on computing answers of GPC queries, i.e., we study the following enumeration problem:

**Problem:** Enumerate answers

**Input:** A property graph \( G \), and a query \( Q \).

**Output:** Enumerate all pairs \( (p, \mu) \in [Q]_G \) without repetitions.

A potential criticism we would like to address is the fact that the path \( p \), witnessing the output mapping \( \mu \), is also returned each time, which can make query answers larger than necessary. The reason that we study the problem like this, however, is that this is what the GQL standard asks for. In our analysis, we will use Turing machines with output tape (in order to enumerate the results), and we will bound the size of the work tape the machine uses. The main result of this section is the following:

**Theorem 12.** The problem **Enumerate answers** can be solved by a Turing machine using exponential space (in \( G \) and \( Q \)). If we consider the query \( Q \) to be fixed (data complexity), then the machine uses only polynomial space.

**Proof sketch.** The basic idea is to enumerate all possible answers \( (p, \mu) \) in increasing length of \( p \), and check, one by one, whether they should be output. If we consider a single pattern with a restrictor on top, e.g., \( Q = \rho \pi \), this approach works as described, and the size of the possible paths (and thus also mappings \( \mu \)), can be bounded by a size that is exponential in the size of \( Q \) and \( G \), and polynomial if we assume \( Q \) to be fixed. For each such answer, we can validate whether it should be output in polynomial space. Notice that once a result is output, we can discard it, and move to the next one. Enumeration stops once an appropriate path length has been reached, and the next mapping \( \mu \) is considered. Joins can then be evaluated by nesting this procedure.

**Theorem 13.** The problem **Enumerate answers** cannot be solved by a Turing machine using polynomial amount of space (in \( G \) and \( Q \)).

### 7 LOOKING AHEAD

In this section, we discuss possible extensions of GPC that would reflect additional features envisioned in GQL and SQL/PQG. In doing so, we also provide two examples of how theoretical research has directly influenced the drafts of the GQL and SQL/PQG standards as they were being written.

**Placement of restrictors.** We imposed strict requirements for placing restrictors: optional shortest followed by optional trail or simple, with at least one of the three present to ensure that the number of returned paths is finite. It is natural to wonder whether restrictors could be mixed arbitrarily, by allowing patterns \( \rho \pi \) where \( \rho \) is one of shortest, trail, and simple. In fact, this was an earlier proposal in the GQL and PQG drafts, which was then significantly modified. To see why, consider the following graph

\[
\begin{aligned}
\begin{tikzpicture}
\node (A) at (0, 0) {$A$};
\node (B) at (1, 0) {$B$};
\node (C) at (2, 0) {$C$};
\node (e1) at (0.5, 0.2) {$e_1$};
\node (e2) at (1.5, 0.2) {$e_2$};
\node (e3) at (2.5, 0.2) {$e_3$};
\draw[->] (A) -- node[above]{$a$} (B);
\draw[->] (B) -- node[above]{$b$} (C);
\draw[->] (C) -- node[above]{$c$} (A);
\end{tikzpicture}
\end{aligned}
\]

(where \( e_1, e_2, \) and \( e_3 \) are edge ids) and the pattern

\[
\begin{aligned}
&\text{trail} \left[ \text{shortest} : (A) \xrightarrow{a,\infty} (B) ; (B) \xrightarrow{\rho,\pi} (C) ; (A) \right].
\end{aligned}
\]
Matching the subpattern outside shortest produces the assignment of \( y \) to the edge \( e_2 \). In the GQL rationale, shortest should restrict query answers in the sense that, out of all the answers to the query, it chooses the one with the shortest witness. If we follow this rationale, then, to keep the entire match a trail, the group variable \( x \) must be assigned the list \( \{ e_1, e_3 \} \). Therefore, counter-intuitively, a shortest match occurring under the scope of trail produces a path that is not shortest between two nodes.

As a result, GQL pattern matching now disallows arbitrary mixing of restrictors. At the same time, it is slightly more permissive than the version presented here: shortest must appear at the top of a pattern, but trail and simple can be mixed freely. Adding this feature is a possible extension of GPC.

Aggregation. As one navigates along a path in a graph, aggregation is a natural feature for computing derived quantities, such as path length. For instance, with \( ( : A ) \sum_{0}^{\infty} ( ; B ) \) looking for paths between \( A \) and \( B \), one could return the total length \( \sum x.\text{length} \) of each matched path. However, adding aggregation is problematic. To see why, consider the simple aggregate \( \# x \) for a group variable \( x \), which counts the number of bindings of that variable. Now assume we extend the language with arithmetic conditions of the form \( t_1 = t_2 \), where \( t_1 \) and \( t_2 \) are terms built from values \( y.k \) and \( \# x \) by means of addition \( + \) and multiplication \( \cdot \). These already pack huge expressive power:

**Proposition 14.** The data complexity of GPC with arithmetic conditions is undecidable.

In view of this, the current approach of GQL is to only allow aggregates in the outputs of queries (no operations on them are permitted). But it is a general open direction to understand how to tame and use the power of aggregates in a graph language.

**Bag semantics.** Relational calculus (first-order logic) is interpreted under set semantics, but SQL uses bag semantics, and so does GQL. In the basic version of GPC we opted for set semantics, following the relational calculus precedent, but it is necessary to study bag semantics as an extension of GPC.

**Nulls and conditionally defined variables.** We have left out the treatment of nulls, assuming that conditions involving non-applicable nulls — i.e., values \( \delta(x,k) \) where the property \( k \) is not defined for \( x \) — evaluate to false. Following SQL (and the current GQL proposals) they would instead evaluate to unknown, leading to many known issues [11]. In addition, one could expand the language with a predicate that checks whether a conditional variable is bound, as done, in fact, in GQL.

**Scoping of variables.** Consider the pattern \( (x) \rightarrow (y) \rightarrow (z) \rightarrow (\theta) \rightarrow (u) \), where \( \theta \) is \( x.k + u.k = y.k + z.k \). To evaluate this compositionally, for the subexpression \( (y) \rightarrow (z) \rightarrow (\theta) \) we need to know the values of \( x.k \) and \( u.k \), which do not occur locally. This necessitates looping over all pairs of nodes in the graph, turning the evaluation into a computationally challenging task. Moreover, if the pattern from \( y \) to \( z \) appears under repetition, the evaluation procedure becomes much less clear. Despite these issues, GQL plans to offer such kind of features, which therefore need to be thoroughly investigated.

**Label expressions.** GQL will offer complex label expressions [13], and these too can be added to the calculus as an extension.

**Enhancing conditions.** We assumed that data values come from one countable infinite set of constants, very much in line with the standard presentations of first-order logic. In reality, of course, data values are typed, and such typing must be taken into account (at the very least, for the study of aggregation, to distinguish numerical properties). Furthermore, one could permit explicit equalities between singleton variables in conditions (currently, such variables implicitly join when they are repeated in a pattern).

**Other language features.** GQL is not limited to pattern matching. Similarly to Cypher, it provides relational operations on tabular representations of answers. In this paper we focused on the abstraction of pattern matching alone. For a follow-up that considers additional features of GQL in a syntax that more closely resembles the actual language, see [15].

**ACKNOWLEDGMENTS**

This work was supported by the following grants: a Leverhulme Trust Research Fellowship; EPSRC grants N023056 and S003800; Agence Nationale de la Recherche projects ANR-21-CE48-0015 (Verigraph), ANR-18-CE40-0031 (QUID), ANR project EQUUS ANR-19-CE48-0019 which is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 431138758; Poland’s National Science Centre grant 2018/30/E/ST6/00042; ANID Millennium Science Initiative Program ICN17_002, and ANID Fondecyt regular project 1221799.

We thank the anonymous reviewers for their constructive feedback. We are also grateful to the members of ISO’s SQL/GQL committee, and especially Fred Zemke, for their comments.

**REFERENCES**


