MGQL: An Executable, Small-Step Semantics of GQL

ADITYA THIMMAIAH, University of Texas at Austin, USA
TONG-NONG LIN, University of Texas at Austin, USA
MILOS GLIGORIC, University of Texas at Austin, USA

Research and development of graph query languages has been gaining traction with the increase in popularity of graph databases, specifically due to the flexible schema and other rich semantic offerings of the latter's most common underlying data model: the property graph. This has culminated in the standardization of the ISO Graph Query Language (GQL) as ISO/IEC 39075 in 2024, the first international standard for property graph-based graph query languages. However, ISO/IEC 39075 codifies its semantics informally across 600+ pages of prose, making it difficult to formally reason about the standard or for a standard-faithful implementation.

Existing formalizations are not adequate because they either: (1) significantly reduce the semantic complexity by omitting bag semantics, schemas, and composite queries on multiple graphs; (2) or significantly reduce the syntactic complexity by only considering isolated fragments such as pattern-matching, leaving the full query pipeline unformalized. Yet it is these semantic–syntactic features that make formalizing GQL non-trivial.

We present MGQL, the first mechanized, small-step operational semantics for a substantial read-only fragment of GQL that is grounded in the ISO/IEC 39075 standard. Our formalization models multi-graph property graphs with mixed edge directionality and supports a large fraction of GQL pattern constructs: quantified paths and edges, directional and undirected matching, label expressions, pattern lists, and composite queries. The semantics is supported by a schema-aware type system that refines variable types via closed-graph schemas, tracks nullability, supports multiple composite query operators, and models quantified-path bindings with list types. We prove that our type system is sound, ensuring an end-to-end guarantee of well-formed queries yielding results that conform to their declared schemas. MGQL provides the first bridge between GQL's informal specification and a mechanized implementation, enabling formal reasoning about correctness.

CCS Concepts: • **Theory of computation → Type theory**; **Operational semantics**; • **Software and its engineering → Semantics**; • **Information systems → Query languages for non-relational engines**.

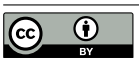
Additional Key Words and Phrases: ISO GQL, Cypher, Graph Query Language, Formal Semantics, Lean, Mechanization

ACM Reference Format:
Aditya Thimmaiah, Tong-Nong Lin, and Milos Gligoric. 2026. MGQL: An Executable, Small-Step Semantics of GQL. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 327 (October 2026), 27 pages. https://doi.org/10.1145/3839459

1 Introduction

Graph databases have become foundational for modern data management and power many applications from social networks and fraud detection to drug discovery and knowledge graphs [1, 27]. The Graph Query Language (GQL) [11, 16], standardized in 2024 as ISO/IEC 39075, is the first international standard of graph query languages that query property graphs [1]. The standard

Authors' Contact Information: Aditya Thimmaiah, University of Texas at Austin, Austin, USA, auditt@utexas.edu; Tong-Nong Lin, University of Texas at Austin, Austin, USA, tong-nong@utexas.edu; Milos Gligoric, University of Texas at Austin, Austin, USA, gligoric@utexas.edu.



ACM 2475-1421/2026/10-ART327
https://doi.org/10.1145/3839459

unified language features from other graph query languages such as Cypher [15], PGQL [30], and G-CORE [2]. It defines the semantics specification which implementations are expected to conform.

However, the semantics is specified informally, spanning more than 600 pages of prose that interleave data flow, typing, query evaluation pipelines, and pattern-matching, etc., making formally reasoning about GQL, such as verifying implementations or proving query equivalences, difficult.

Existing formalizations do not address this adequately. Francis et al. [13] distilled GQL's pattern matching into a Graph Pattern Calculus (GPC) with typing rules and a denotational semantics, and Gheerbrant et al. [16] defined Core GQL and Core PGQ to study its expressive power, but both leave out graph schemas [4] and bag semantics. Ye et al. [31] formalized a gradually typed calculus for GQL path patterns in isolation from the rest of the language, implementing it in Python. No mechanized semantics exists for a fragment of GQL rich enough to model the interaction between graph schemas, bag semantics, and composite queries, etc., some of the key unique features of GQL.

We present MGQL, the first mechanized, executable, small-step operational semantics for a large read-only fragment of GQL, together with a schema-aware type system and a type soundness theorem, all machine-checked in Lean4 [9]; providing a foundation for rigorous formal reasoning.

We handle the complexity of GQL's semantics by *stratifying* our static typing rules and small-step operational semantics into compositional layers—expressions, patterns, and queries. This allows us to capture the key non-trivial features of GQL. First, graph element types in GQL are graph-scoped and refined by graph schemas, allowing for label and property constraints to enable statically detecting patterns that cannot match, but requiring a typing system that propagates refined types across the query pipeline. Second, GQL's three-valued semantics identifies null with the truth value *Unknown*, requiring every operator to participate in Kleene-style null propagation and every type to track nullability. Third, GQL's *path modes* and *match modes* impose additional constraints on pattern matching such as the `TRAIL` path mode which constrains a pattern-match by forbidding repeated edges, thus breaking naïve path composition. We handle this by introducing auxiliary *execution constructs* besides those specified in the standard, which carry the edges visited during pattern-matching alongside the matched elements, making it trivial to enforce edge-disjointness. Fourth, quantified path patterns iterate over graph structure and bind their variables to sequences rather than single elements. We compute their repetitions with a frontier. Finally, composite queries combine bags whose records carry the same attributes but not necessarily the same types, yielding heterogeneous unions that the type system must track to formalize well-formedness of queries.

We make the following contributions:

- We define a **formal calculus** (§2.2) for a rich read-only fragment of GQL, covering graph resolution, pattern matching, filtering, projection, and composite queries over multiple graphs.
- We develop the first **schema-aware type system** (§3) for GQL, supporting graph-scoped refinement, nullability tracking, heterogeneous unions, and list-typed quantified path bindings.
- We give the first **small-step operational semantics** (§5) for GQL, capturing `TRAIL`-aware pattern matching, quantified patterns, nulls, and composite queries; all under bag semantics.
- We prove a **layered type soundness theorem** (§6) guaranteeing that well-formed queries (on complete evaluation) produce results that conform to the types assigned to them statically.
- We **mechanize** (§7) our semantics in Lean4, and validate it on queries of the Linked Data Benchmark Council's (LDBC) Social Network Benchmark (SNB) Interactive v2 workload [22].

All references to the GQL standard are with respect to the ISO/IEC 39075 First Edition 2024, which we henceforth refer to as just ISO. We also abbreviate normative *syntax rules* from the standard as SR when referring to them. MGQL is available at https://github.com/EngineeringSoftware/mgql.

2 Preliminaries

We now formally introduce the two core components our work is built around: the property graph data model (§2.1) and the GQL graph query language (§2.2) that operates on them.

Notation 2.1 (Metafunctions and Shorthands). We use the metafunctions $\mathbf{dom}(\cdot)$ and $\mathbf{supp}(\cdot)$ to denote the *domain* and the *support* of a function; the uppercase calligraphic letters (e.g., $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$) as well as accents (e.g., $\overline{L}$, $\overline{\zeta}$) to denote sets; and when convenient, the corresponding unaccented symbols denote elements ranging over those sets (e.g., $\zeta \in \overline{\zeta}$). For a set $\mathcal{S}$ and $m \in \mathbb{N}$, $\mathcal{P}_{\leq m}(\mathcal{S})$ denotes the set of subsets of $\mathcal{S}$ whose cardinality is at most m, i.e., $\mathcal{P}_{\leq m}(\mathcal{S}) \triangleq \{ C \subseteq \mathcal{S} \mid |C| \leq m \}$. The metafunction $\boldsymbol{\pi_k}(\cdot)$ for a natural number k, denotes access to the k^{th} component of a tuple. When applied to a set or a bag of tuples, it lifts pointwise and returns the set or bag of the k^{th} components of those tuples, respectively. □

2.1 The Property Graph Data Model

The property graph data model [1, 26] is richer than the edge-labeled graphs typically used in theoretical work on regular path queries [5]: nodes and edges carry *labels* describing their domain role, *properties* stored as key–value pairs, and *directionality* markers that distinguish directed from undirected relationships. This additional structure makes GQL's type system nontrivial with label expressions, property constraints, join conditions, etc., all inspecting graph structure absent in simpler models and abstractions. We formalize it in full without reducing to a minimal abstraction.

Figure 1 shows an example concrete property graph instance. The graph contains eight nodes carrying one of three node labels: five `Person` nodes {a, c, e, o, z}, two `Company` nodes {x, u}, and one `Project` node {s}. Edges carry one of five labels: `KNOWS`, `WORKS`, `LEADS`, `FUNDS`, and `INVESTS`, forming a multigraph with parallel edges and cycles. The graph is mixed and contains directed edges (e.g., the `WORKS` edge in o −WORKS→ u) as well as undirected edges (e.g., the `KNOWS` edge in e −KNOWS− z). Edge labels denote the semantic role of the relationship connecting two nodes, and two nodes may be connected by more than one edge (multigraphs). Many nodes also carry property maps, like the `Project` node s: {`name` ↦ "`Atlas`", `projectID` ↦ 451}. These properties illustrate the key feature of property graphs: nodes and edges may carry arbitrary and finite (possibly empty) key–value maps whose values range over primitive domains such as integers, strings, and booleans; they are used for capturing the attributes of the data being modeled.

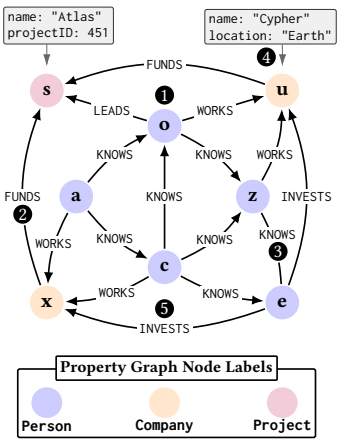


Fig. 1. An example of a property graph instance that contains: nodes (❶), directed edges (❷), undirected edges (❸), node property maps (❹), and labeled edges (❺).

Graph Instance. We assume a multi-sorted universe of values $\mathscr{U}$ given as the disjoint union of sub-universes of the types $\mathcal{T}_0 \triangleq \mathbb{Z} \mid \mathbb{S} \mid \mathbb{B}$; with $\mathbb{Z}$, $\mathbb{S}$, and $\mathbb{B}$ denoting the universes of integers, strings, and booleans, respectively. We also assume an infinite universe of named identifiers $\mathcal{A}$. A *property map* $\Pi : \mathcal{A} \rightharpoonup \mathscr{U}$ is a finite partial map from elements (*keys*) in $\mathcal{A}$ to those (*values*) in $\mathscr{U}$.

Definition 2.1 (Property Graph Instance). A *property graph* $\mathcal{G}$ is a tuple $(\mathcal{N}, \mathcal{E}, \Delta, \Theta, \Lambda, \Xi)$ where:

(1) $\mathcal{N}$ and $\mathcal{E}$ are finite sets of *nodes* and *edges*, respectively.
(2) $\Delta : \mathcal{E} \to \{\circlearrowright, \bowtie\}$, gives the *directionality* of edges; whether directed ($\circlearrowright$) or undirected ($\bowtie$).

(3) $\mathcal{E} = \mathcal{E}^{\circlearrowright} \sqcup \mathcal{E}^{\bowtie}$, the set of edges $\mathcal{E}$ can be partitioned into sets $\mathcal{E}^{\circlearrowright}$ and $\mathcal{E}^{\bowtie}$ based on directionality.
(4) $\Theta : \mathcal{E}^{\circlearrowright} \sqcup \mathcal{E}^{\bowtie} \to (\mathcal{N} \times \mathcal{N}) \sqcup \binom{\mathcal{N}}{2}$, maps edges to their endpoint nodes. For $e \in \mathcal{E}^{\circlearrowright}$, $\Theta(e) = (n_1, n_2) \in \mathcal{N} \times \mathcal{N}$ is an *ordered pair* with n_1 and n_2 referred to as its *source* and *target* nodes. For $e \in \mathcal{E}^{\bowtie}$, $\Theta(e) = \{n_1, n_2\} \in \binom{\mathcal{N}}{2}$ is an *unordered pair*. Self-loops are permitted, i.e., $n_1 = n_2$.
(5) $\Lambda : \mathcal{N} \cup \mathcal{E} \to \mathcal{P}_{\leq m}(\mathcal{A})$, $m \in \mathbb{N}$; maps nodes/edges to finite (maybe empty) sets of *labels* from $\mathcal{A}$.
(6) $\Xi : \mathcal{N} \cup \mathcal{E} \to \Pi$, maps nodes/edges to their properties. Π may be empty, i.e., $\mathbf{dom}(\Xi(\cdot)) = \varnothing$.

Graph Schemas. The GQL standard permits property graphs to be associated with *graph schemas* [4], which constrain the admissible labels and properties on nodes and edges (ISO §4.13.2).

Definition 2.2 (Property, Node, Edge, and Graph Schemas). Let $\overline{L} \subset \mathcal{A}$ be a set of labels.

[Φ]. A *property schema* (ISO §4.13.2.5) $\Phi : \mathcal{A} \rightharpoonup \mathcal{T}_0$ is a finite partial, mapping keys to scalar types.
[ζ]. A *node schema* (ISO §4.13.2.3) $\zeta \triangleq (\overline{L}, \Phi)$ is a pair of a set of labels $\overline{L}$ and a property schema Φ.
[ξ]. An *edge schema* (ISO §4.13.2.4) $\xi \triangleq (\overline{L}, \Phi, \zeta_1, \zeta_2, \delta)$ is a tuple of a set of labels $\overline{L}$, a property schema Φ, its endpoint node schemas ζ_1 and ζ_2, and edge directionality $\delta \in \{\circlearrowright, \bowtie\}$.
- For $\delta = \circlearrowright$, ζ_1 and ζ_2 correspond to the source and target node schemas, respectively.
- For $\delta = \bowtie$, ζ_1 and ζ_2 are unordered endpoint schemas.

[Ψ]. A *graph schema* (ISO §4.13.2.2) $\Psi \triangleq (\overline{\zeta}, \overline{\xi})$ is a pair of finite sets of node $\overline{\zeta}$ and edge $\overline{\xi}$ schemas.

Definition 2.3 (Conformance Relations). Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Lambda, \Theta, \Lambda, \Xi)$ be a property graph, $\Psi = (\overline{\zeta}, \overline{\xi})$ a graph schema, and $\Phi : \mathcal{A} \rightharpoonup \mathcal{T}_0$ a property schema. Suppose $\zeta \in \overline{\zeta}$ and $\xi \in \overline{\xi}$.

[$\Pi \models \Phi$]. A property map Π *conforms* to Φ when $\mathbf{dom}(\Pi) = \mathbf{dom}(\Phi) \wedge \forall k \in \mathbf{dom}(\Pi).\Pi(k) \in \mathscr{U}_{\Phi(k)}$.
[$n \models^{N}_{\mathcal{G}} \zeta$]. A node $n \in \mathcal{N}$ *conforms* to a node schema $\zeta = (\overline{L}, \Phi)$, when $\Lambda(n) = \overline{L}$ and $\Xi(n) \models \Phi$.
[$e \models^{E}_{\mathcal{G}} \xi$]. An edge $e \in \mathcal{E}$ *conforms* to an edge schema $\xi = (\overline{L}, \Phi, \zeta_1, \zeta_2, \delta)$ when $\Lambda(e) = \overline{L}$, $\Xi(e) \models \Phi$, and:
- $e \in \mathcal{E}^{\circlearrowright}$ for $\delta = \circlearrowright$ with $n_1 \models^{N}_{\mathcal{G}} \zeta_1$ and $n_2 \models^{N}_{\mathcal{G}} \zeta_2$ where $\Theta(e) = (n_1, n_2)$; or
- $e \in \mathcal{E}^{\bowtie}$ for $\delta = \bowtie$ with $\exists i, j \in \{1, 2\}\ .i \neq j.n_i \models^{N}_{\mathcal{G}} \zeta_1$ and $n_j \models^{N}_{\mathcal{G}} \zeta_2$ where $\Theta(e) = \{n_1, n_2\}$.

[$\mathcal{G} \models \Psi$]. The graph $\mathcal{G}$ *conforms* to Ψ when $\forall n \in \mathcal{N}.\ \exists \zeta \in \overline{\zeta}.\ n \models^{N}_{\mathcal{G}} \zeta$ and $\forall e \in \mathcal{E}.\ \exists \xi \in \overline{\xi}.\ e \models^{E}_{\mathcal{G}} \xi$.

Definition 2.4 (Closed and Open Property Graphs). Let $\overline{\Psi}$ be a set of graph schemas and $\mathcal{G}$ a property graph. $\mathcal{G}$ is defined as *closed* or *open* under $\overline{\Psi}$ based on whether $\exists \Psi \in \overline{\Psi}.\ \mathcal{G} \models \Psi$ or $\nexists \Psi \in \overline{\Psi}.\ \mathcal{G} \models \Psi$, respectively, i.e., closed graphs conform to atleast one schema, while open graphs conform to none.

The ISO standard refers to the node, edge, and graph schemas from Definition 2.2 as their respective *types*. This alias is evident from the conformance relations (Definition 2.3), where a graph or a graph element, i.e., node or edge, *conforms* to a schema only when it satisfies the schema's constraints on all of its semantic attributes, i.e., labels, properties, directionality, etc., which can be likened to fields. Therefore, when a graph conforms to a schema, i.e., closed graphs, the schema can be leveraged to type GQL queries operating on that graph more precisely, i.e., assign narrower types. We design our type system (§3) to exploit the type information inferrable from the schemas.

2.2 GQL

GQL [14, 20] is the first international standard for a property-graph query language, standardized in 2024 as ISO/IEC 39075; and aimed at unifying ideas and concepts from other popular graph query languages such as Cypher [15], PGQL [30], and G-CORE [2], etc., into a single and vendor-neutral specification. Similar to relational query languages like SQL, a GQL query specifies *what* to retrieve from data sources, and optionally transform them before retrieving; while the query

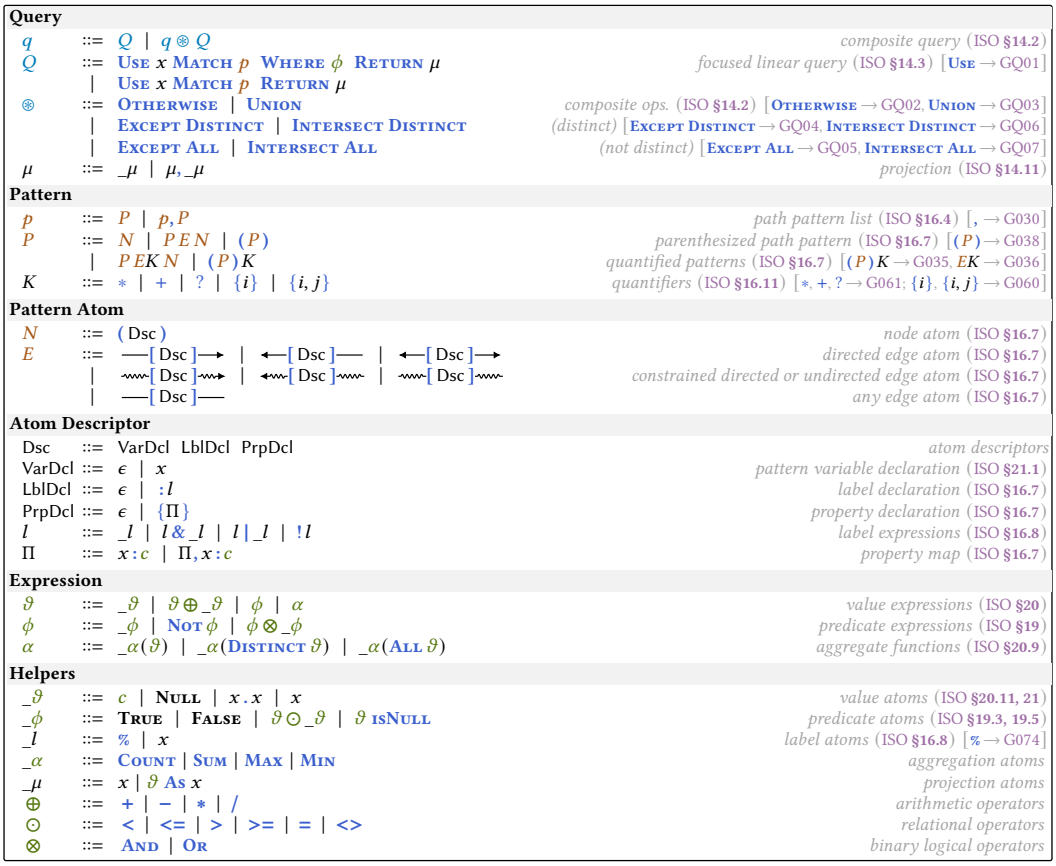


Fig. 2. The formalized GQL calculus with the relevant ISO sections and optional features (GXXX), described in gray. $x \in \mathcal{A}$; $i, j \in \mathbb{N}$; $\mathscr{U}_{\mathbb{B}} = \{\textsc{True}, \textsc{False}\}$; and $c \in \mathscr{U}_{\tau}$, $\tau \in \mathcal{T}_0$. We use the metavariable A to denote N and E.

engine determines *how* to evaluate the query. Broadly, data sources in query languages can be divided into constructed sources and stored sources. Property graphs serve as stored data sources in graph query languages, while *base tables* [19] are an example of stored sources in SQL. Since base tables are highly structured data sources with rigid schemas, a fragment supporting just column projections is often sufficient purely for data retrieval in SQL lineage of languages. Property graphs on the other hand have far more flexible schemas, and so graph query languages include a rich graph pattern matching fragment specifically for data retrieval. The fragment for the optional data transformations before retrieving, largely overlap between relational and graph query languages.

The interplay between the graph pattern matching fragment for data retrieval and the remaining largely relational fragment for data transformations, make GQL's semantics non-trivial. The semantics is further complicated by GQL allowing querying multiple graphs within a single query; with different ways of combining their results, via composite queries and operators. We formalize a GQL fragment that captures the difficult semantic challenges arising from the interplay.

Formalized GQL Fragment. Figure 2 shows the GQL fragment we formalize. A GQL query q is either a focused linear query Q, i.e., a query operating on a single graph instance, without any composite query operators; or a composite query $q \circledast Q$ that combines the results of two or more sub-queries using composite query operators $\circledast \in \{\textsc{Otherwise}, \textsc{Union}, \textsc{Except}, \textsc{Intersect}\}$.

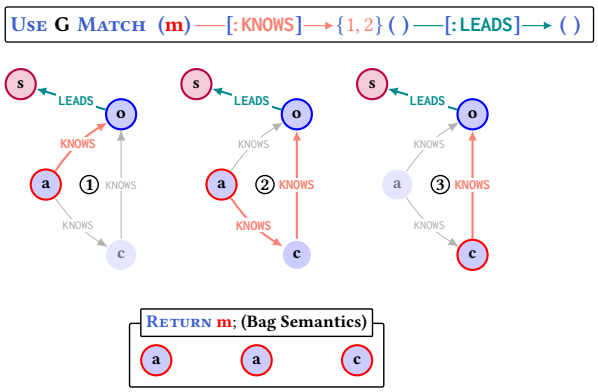


Fig. 3. An example of graph pattern matching with GQL.

A focused linear query Q first declares the graph being queried using the **USE** x clause, where $x \in \mathcal{A}$ refers to the name assigned to the graph instance in the database's catalog. We use G specifically to refer to graph names, i.e., **USE** G selects the graph instance named G in the catalog as the graph being queried (also called the working graph). Declaration of the working graph is followed by a **MATCH** p clause which is the heart of GQL's graph pattern matching fragment. It specifies a pattern list p. Next, an optional **WHERE** ϕ clause for filtering the matched bindings using the predicate expression ϕ. Finally, a **RETURN** μ clause for returning the query result after projecting the filtered matched bindings via the list of projections μ.

A pattern list p is built from path patterns P composed via conjunction p, P: a comma-separated list requiring *all* patterns to match. A path pattern P is a sequence of node atoms N connected by—only directed ($\text{—}\cdot\text{→}$, $\text{←}\cdot\text{—}$, $\text{←}\cdot\text{→}$), right directed or undirected ($\sim\cdot\sim\text{→}$), left directed or undirected ($\text{←}\sim\cdot\sim$), only undirected ($\sim\cdot\sim$), and unconstrained ($\text{—}\cdot\text{—}$)—edge atoms E. Edges and patterns can be quantified by quantifiers K whose semantics parallels that of regular-expression repetition. The quantifier K includes: Kleene star ($*$), Kleene plus ($+$), optionality ($?$), exact repetition ($\{i\}$), and bounded repetition ($\{i, j\}$), where $i, j \in \mathbb{N}$ and $i \leq j$. Node atoms N may declare binding variables x, specify a label expression l, and impose property constraints via a property map PrpDcl. Same holds for the edge atoms E. Label expressions l are formed from label names ($L \in \mathcal{A}$) and the wildcard ($\%$) under: conjunction (&), disjunction (|), and negation (!). This allows for fine-grained control over which graph elements a pattern atom matches.

Value expressions ϑ include constants, variables, property accesses $x.x$, arithmetic, and aggregate function applications α. Predicates ϕ are formed from Boolean constants **TRUE**/**FALSE**, relational operations $\vartheta \odot \vartheta$, the null test ϑ **ISNULL**, and the logical connectives (**AND**, **OR**, **NOT**). Aggregate functions **COUNT**, **SUM**, **MAX**, and **MIN** may be applied with a **DISTINCT** or **ALL** (default) qualifier, which decides if the aggregating operation should be performed under set or bag semantics.

Execution Constructs. A GQL query executes within a *database world* $\Sigma = (\Omega, \Upsilon)$. The *graph catalog* $\Omega : \mathcal{A} \rightharpoonup \mathcal{G}$ is a fixed collection that resolves graph names ($G \in \mathcal{A}$) to property graph instances (ISO §4.2.5). The *schema mapper* $\Upsilon : \mathcal{A} \rightharpoonup \Psi$ associates each graph name with a graph schema (if defined). Schema information is available only for closed graphs. The *working graph site* G designates the graph currently in scope and is selected from the catalog via the **USE** clause. All pattern matching in the query after graph selection operates against that graph (ISO §4.7.5).

We formally define GQL's execution constructs when formalizing the semantics (§5). Here, we introduce them informally for contextual reasons. A *record* is a finite set of fields, each pairing a variable name with a runtime value (§5.1)—a scalar, a graph-element identity, or the null value (ISO §4.15.4). A *binding table* is a bag of records having compatible schema (Definition 3.3). It serves as the primary execution construct in query evaluation pipeline, by holding intermediary results such as the matches found by graph pattern matching, and constituting the execution result returned after query evaluation (ISO §4.3.6). Every clause in a query transforms a binding table: MATCH produces one, WHERE filters it, and RETURN projects it into the query's result.

Evaluating MATCH p against the working graph enumerates all graph homomorphisms $h : p \to \mathcal{G}$ that satisfy the structural, label, and property constraints of p. Each such homomorphism yields a record that binds the pattern variables to the matched graph elements, and the collection of all such records forms the binding table produced by the MATCH clause. Pattern conjunction combines binding tables via a join on shared variables.

The WHERE clause retains only those records for which the predicate ϕ evaluates to true, and the RETURN clause maps each surviving record to its projected expressions, producing the output binding table of the query. Composite queries compose queries by combining their binding tables under different composite query operators: UNION—union, INTERSECT—intersection, EXCEPT—difference, and OTHERWISE returns the left operand's table when non-empty otherwise the right's.

Example 2.1 (GQL Graph Pattern Matching). Consider the example GQL query shown in Figure 3:

USE G MATCH (**m**) −[:KNOWS]→ {1, 2}( ) −[:LEADS]→ ( ) RETURN **m**;

This query has the shape MATCH P RETURN μ, from the core calculus in Figure 2. The projection item is $\mu = \mathbf{m}$ and a single path pattern $P =$ (**m**) −[:KNOWS]→ {1, 2}( ) −[:LEADS]→ ( ). The pattern P requires a node **m** from which there exists a path of one or two KNOWS edges to some intermediate node, followed by a single outgoing LEADS edge to a final node. Evaluation against the property graph instance $\mathcal{G}$ in Figure 1 proceeds by enumerating all graph homomorphisms $h : P \to \mathcal{G}$ satisfying the path constraint, yielding three records in the binding table:

① **m** = **a**, via the path **a** −KNOWS→ **o** −LEADS→ **s**.
② **m** = **a**, via the longer path **a** −KNOWS→ **c** −KNOWS→ **o** −LEADS→ **s** from quantifier {1, 2}.
③ **m** = **c**, via the path **c** −KNOWS→ **o** −LEADS→ **s**.

Since no WHERE clause is present, all three records survive filtering. The RETURN **m** clause then projects each record onto the variable **m**, and under bag semantics the result is the bag {{ **a**, **a**, **c** }} containing one entry per homomorphism, including the duplicate binding of **m** to **a**. □

Metafunctions for Parsing Patterns. Our formalization uses several metafunctions for parsing GQL query patterns as shown in Figure 4. **var**(·), **lbl**(·), and **prp**(·) return the binding variable, label expression, and property map of a pattern atom (node or edge). We interpret the property map Π returned by **prp**(·) as a partial, mapping property names to values. **lo**(·) returns the lower bound of a quantifier. **tail**(·) returns the binding variable of a pattern's rightmost node atom. $\mathbf{lo}^{\#\mathbf{E}}(\cdot)$ returns the minimum number of edge atoms in a pattern, while $\mathbf{lo}^{\#\mathbf{N}}(\cdot)$ returns the minimum number of node atoms in a pattern. Both account for quantifiers. Finally, **ornt**(·) maps an edge atom based on its direction, to a set of edge *orientations*: left directed (←), undirected (~), and right directed (→); while **dir**(·) maps orientations to directionalities: directed (↻) or undirected (⊠).

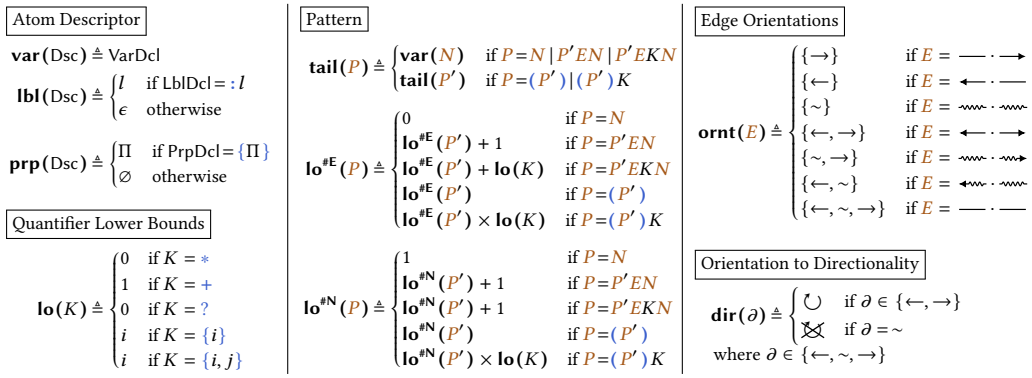


Fig. 4. Metafunctions for parsing GQL's graph pattern matching fragment. *Atom Descriptor* metafunctions **var**, **lbl** and **prp**; lift to pattern atoms N and E via their descriptors (Dsc). Only the quantifier lower bounds $\mathbf{lo}(K)$ are given since interpreting upper bounds requires a specific property graph instance (for `TRAIL` mode).

3 Type System

The graph schema support intrinsic to GQL's standard allows for more precise typing. We design our type system to exploit the availability of graph schemas to propagate their graph-structural invariants through query patterns into query results. Broadly, our design includes four features of the GQL specification, some of which are absent from simpler formalisms [11, 13]: (1) *graph scoping*—elements of different graphs cannot be conflated with each other; (2) *schema refinement*—closed graphs carry schemas that constrain semantic attributes of their elements allowing for more precise typing; (3) *nullability*—missing properties and failed matches produce null values which obey three-valued logic (3VL); (4) and *heterogeneous unions*—composite queries and property accesses produce bindings whose runtime type varies across records within the same binding table.

$$
\begin{array}{lll}
\mathcal{T}_0 ::= \mathbb{Z} \ \textit{integer} & \mathcal{T}_1 ::= \mathcal{T}_0 \ \textit{scalars} & \mathcal{T}_2 ::= \mathcal{T}_1 \ \textit{value types} \\
\quad \mid \mathbb{B} \ \textit{boolean} & \quad \mid \mathsf{N}\langle - \rangle \ \textit{node} & \quad \mid \mathbf{Any} \ \textit{top type} \\
\quad \mid \mathbb{S} \ \textit{string} & \quad \mid \mathsf{N}\langle -, - \rangle \ \textit{node} & \quad \mid \bot \ \textit{bottom type} \\
 & \quad \mid \mathsf{E}\langle - \rangle \ \textit{edge} & \quad \mid \mathsf{?} \ \textit{null type} \\
 & \quad \mid \mathsf{E}\langle -, - \rangle \ \textit{edge} & \quad \mid \mathcal{T}_2 \cup \mathcal{T}_2 \mid \mathbf{List}\ \mathcal{T}_2 \mid \mathcal{T}_2^{\perp} \ \textit{type formers}
\end{array}
$$

Our type system is organized into three tiers. $\mathcal{T}_0$ gives *scalar types* (property-value primitives). $\mathcal{T}_1$ extends scalars with *graph-scoped element types*—node and edge identity types indexed by a graph site and optionally refined by a schema entry. $\mathcal{T}_2$ gives the full *value type* language: element types $\mathcal{T}_1$, the top type **Any**, the bottom type $\bot$, the null type **?**, and is closed under the types formed from the union, the list, and the empty $(\cdot)^{\perp}$ formers. Unless otherwise stated, we use τ to range over $\mathcal{T}_2$.

We now motivate our types and type formers, starting with graph-indexed and schema-refined types (§3.1), followed by the union former (§3.2), then the list former (§3.3), and finally, the null type and the empty former (§3.4). We conclude with the subtyping rules for our type system (§3.5).

3.1 Graph-Indexed and Schema-Refined Types

GQL supports composite queries where the individual queries may query different property graphs. However, graph elements are inherently *graph-local*, i.e., nodes and edges of one property graph

cannot be conflated with another. We prevent cross-graph element conflation through *graph-indexed types*: for a graph site G, the types $\mathbf{N}\langle \mathrm{G} \rangle$ and $\mathbf{E}\langle \mathrm{G} \rangle$ classify node and edge identities scoped to G.

For graph sites referring to closed graphs (Definition 2.4), the type system refines graph-indexed types with schema entries. The schema-refined types $\mathbf{N}\langle \mathrm{G}, \zeta \rangle$ and $\mathbf{E}\langle \mathrm{G}, \xi \rangle$ classify elements whose labels, properties, etc., conform (Definition 2.3) to their respective schema entries. Refined types carry strictly more static information, enabling more precise type assignments and propagations.

3.2 GQL's Dynamic Union Types: Union Type Former

Example 3.1 (Heterogeneous Unions). Consider the composite GQL query,

$$\underbrace{\textsc{Use}\ \mathrm{G}\ \textsc{Match}\ (v_1{:}\mathtt{COMPANY})\ \textsc{Return}\ v_1\ \textsc{As}\ \mathbf{x}}_{\text{Returns a binding table with column } \mathbf{x} \text{ typed as } \mathbf{N}\langle \mathrm{G}, \zeta_1 \rangle}\ \textsc{Union}\ \underbrace{\textsc{Use}\ \mathrm{G}\ \textsc{Match}\ (v_2{:}\mathtt{PROJECT})\ \textsc{Return}\ v_2\ \textsc{As}\ \mathbf{x}}_{\text{Returns a binding table with column } \mathbf{x} \text{ typed as } \mathbf{N}\langle \mathrm{G}, \zeta_2 \rangle}$$

evaluated against the property graph (assuming closed) from Figure 1. Suppose the two individual focused linear queries assign different types ($\mathbf{N}\langle \mathrm{G}, \zeta_1 \rangle$ and $\mathbf{N}\langle \mathrm{G}, \zeta_2 \rangle$) to the column $\mathbf{x}$ in their respective binding tables; the combined binding table must classify $\mathbf{x}$ as inhabiting *either* type. □

Example 3.1 illustrates the need for supporting heterogeneous type unions due to GQL's composite queries. Similar situations may arise from property accesses. We model heterogeneous unions by closing our type system under the union former. For any two types τ_1 and τ_2, the dynamic union $\tau_1 \cup \tau_2$ is a static type that admits values whose type inhabits either branch, i.e., the union former does not introduce a new kind of runtime value, every inhabiting value belongs to either τ_1 or τ_2.

Notation 3.1 (Schema-Refined Union Types). Consider a graph site G with graph schema $(\overline{\zeta}, \overline{\xi})$, and some *non-empty* set of node $\overline{\zeta}' \subseteq \overline{\zeta}$ and edge $\overline{\xi}' \subseteq \overline{\xi}$ schemas. We denote the node and the edge schema refined types formed from these sets of node and edge schemas, closed under the union type former, using the syntactic sugars: $\mathbf{N}\langle \mathrm{G}, \overline{\zeta}' \rangle \triangleq \bigcup_{\zeta \in \overline{\zeta}'} \mathbf{N}\langle \mathrm{G}, \zeta \rangle$ and $\mathbf{E}\langle \mathrm{G}, \overline{\xi}' \rangle \triangleq \bigcup_{\xi \in \overline{\xi}'} \mathbf{E}\langle \mathrm{G}, \xi \rangle$. □

3.3 List Types and Reference Modes

Binding variables declared inside quantified path patterns in GQL can contribute multiple bindings to a single match (ISO §4.11.5). Consider Example 2.1 with the quantified KNOWS segment annotated with edge variable $\mathbf{x}$: $(\mathbf{m})$ —[$\mathbf{x}$:KNOWS]→{1, 2}( ) —[:LEADS]→( ). The variable $\mathbf{m}$ (outside the quantifier) remains *singleton*, i.e., each match binds it to exactly one node. The variable $\mathbf{x}$ (inside the quantifier $\{1, 2\}$) is *group-reference*, i.e., contributes multiple bindings per match: a one-hop match yields [—KNOWS→], a two-hop match yields [—KNOWS→, —KNOWS→]. The binding exposed outside the quantifier is not a single edge but the finite list of all edges matched by that segment.

We model this statically via $\mathbf{List}\ \tau$: if a binding variable has type τ inside the quantifier, crossing the quantifier boundary lifts it to $\mathbf{List}\ \tau$. List types thus serve as the static representation of group-reference bindings. In the example above, $\mathbf{x}$ is assigned $\mathbf{List}\ \mathbf{E}\langle \mathrm{G} \rangle$ outside the quantified segment.

Definition 3.1 (Component Type). The *component type* $\lfloor \cdot \rfloor$ strips the outermost **List** wrapper, i.e., $\lfloor \mathbf{List}\ \tau \rfloor = \tau$. The component type of a non-list type is the type itself.

3.4 GQL's Immaterial Types: Null and Empty

Null values are pervasive in GQL, as in SQL [17]; arising from missing properties, optional or failed matches, and undefined intermediate results, etc. The null type **?** is inhabited solely by the value **Null**. We use the syntactic sugar $\tau?$ to denote the static type $\tau \cup ?$ formed using the union former.

GQL's standard specifies an empty type $\bot$ which no value inhabits, and motivates it as an opportunity to embed richer statically inferrable information such as knowledge that a binding does not produce any runtime values (e.g., failed matches). The empty type is the bottom type.

We retain GQL's empty type $\bot$, and use it as the bottom type of our type system, but in addition, we also introduce the empty type former $\tau^{\bot}$ to denote a binding that is statically known to produce no runtime value while retaining the underlying type τ as an annotation (GQL's bare empty type $\bot$ erases that information). We motivate this design choice using the Example 3.2 given below.

Example 3.2 (Empty Former). Consider the query with patterns statically known to fail matching.

```
USE G MATCH (x :COMPANY), (x) -[ :KNOWS]-> ( ) RETURN x        Type Check Pass ✓
```

Typing **x** as $\mathsf{N}\langle\mathsf{G}\rangle$ preserves the base type $\mathsf{N}\langle\mathsf{G}\rangle$ but loses the emptiness guarantee, while typing it as $\bot$ captures emptiness but erases the base type $\mathsf{N}\langle\mathsf{G}\rangle$. To understand why erasing the base type is problematic, consider a small change to the query where **x** binds to an edge in the right pattern:

```
USE G MATCH (x :COMPANY), ( x ) -[ x :KNOWS]-> ( ) RETURN x    Type Check Fail ×
                          ---      +++
```

This query must be rejected since **x** cannot bind both a node and an edge at the same depth of graph pattern matching. But typing **x** as $\bot$ erases the base type ($\mathsf{N}\langle\mathsf{G}\rangle$ in the left pattern vs. $\mathsf{E}\langle\mathsf{G}\rangle$ in the right) required for type checking to detect the inconsistency and reject the query. □

The empty type former $(\cdot)^{\bot}$ resolves the problem in Example 3.2 by encoding emptiness while preserving the underlying type as an annotation. This allows emptiness to propagate without sacrificing the ability to reject ill-formed queries. We restrict $(\cdot)^{\bot}$ to the element types $\mathsf{N}\langle-\rangle$ and $\mathsf{E}\langle-\rangle$. We use $\mathsf{N}\langle-,\varnothing\rangle$ and $\mathsf{E}\langle-,\varnothing\rangle$ as syntactic sugars to denote $\mathsf{N}\langle\mathsf{G}\rangle^{\bot}$ and $\mathsf{E}\langle\mathsf{G}\rangle^{\bot}$, respectively.

3.5 Subtyping

The subtyping relation $\tau_1 <: \tau_2$ is the least reflexive and transitive relation closed under:

$$\frac{}{\tau <: \tau}\ \text{S-Reflexive} \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ \text{S-Transitive} \qquad \frac{}{\tau <: \mathsf{Any}}\ \text{S-Any} \qquad \frac{}{\bot <: \tau}\ \text{S-Empty} \qquad \frac{\tau_1 <: \tau_2}{\mathsf{List}\ \tau_1 <: \mathsf{List}\ \tau_2}\ \text{S-List}$$

$$\frac{}{\tau_1 <: \tau_1 \cup \tau_2}\ \text{S-Union-Left} \qquad \frac{}{\tau_2 <: \tau_1 \cup \tau_2}\ \text{S-Union-Right} \qquad \frac{\tau_1 <: \tau \quad \tau_2 <: \tau}{\tau_1 \cup \tau_2 <: \tau}\ \text{S-Union} \qquad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \cup \tau_2 <: \tau_1' \cup \tau_2'}\ \text{S-Union-Congruence}$$

$$\frac{}{\mathsf{N}\langle\mathsf{G},-\rangle <: \mathsf{N}\langle\mathsf{G}\rangle}\ \text{S-Ref-Node} \qquad \frac{}{\mathsf{N}\langle\mathsf{G}\rangle^{\bot} <: \mathsf{N}\langle\mathsf{G},-\rangle}\ \text{S-Ref-Node-Empty} \qquad \frac{}{\mathsf{E}\langle\mathsf{G},-\rangle <: \mathsf{E}\langle\mathsf{G}\rangle}\ \text{S-Ref-Edge} \qquad \frac{}{\mathsf{E}\langle\mathsf{G}\rangle^{\bot} <: \mathsf{E}\langle\mathsf{G},-\rangle}\ \text{S-Ref-Edge-Empty}$$

S-Reflexive/S-Transitive makes the subtyping relation a preorder. S-Any/S-Empty bound the type system with top **Any** and bottom $\bot$ types. S-List lifts types to their list types. The union rules—S-Union-Left, S-Union-Right, S-Union, and S-Union-Congruence—make $\tau_1 \cup \tau_2$ operate as least-upper-bound. The schema rules—S-Ref-Node and S-Ref-Node-Empty—connect graph-indexed and schema-refined node element types while bounding this sublattice with the annotated empty type $\mathsf{N}\langle\mathsf{G}\rangle^{\bot}$ at the bottom. S-Ref-Edge and S-Ref-Edge-Empty do the same for edge types.

Definition 3.2 (Type Intersection). Let τ_1 and τ_2 be any two types. We define their intersection, written $\tau_1 \cap \tau_2$, as the least reflexive and transitive relation closed under the following judgements:

$$\textsc{Intersect-Left}\ \frac{\tau_1 <: \tau_2}{\tau_1 \cap \tau_2 : \tau_1} \qquad \textsc{Intersect-Right}\ \frac{\tau_2 <: \tau_1}{\tau_1 \cap \tau_2 : \tau_2} \qquad \textsc{Intersect}\ \frac{\tau <: \tau_1 \quad \tau <: \tau_2 \quad \tau_1 \not<: \tau_2 \quad \tau_2 \not<: \tau_1}{\tau_1 \cap \tau_2 : \tau}$$

Lemma 3.1 (Absorption). (1) $\tau \cup \mathbf{Any} = \mathbf{Any}$, (2) $\tau \cup \bot = \tau$, (3) $\tau \cap \mathbf{Any} = \tau$, and (4) $\tau \cap \bot = \bot$.

Lemma 3.2 (Schema-Refined Type Bounds). Consider a graph site G with graph schema $(\overline{\zeta}, \overline{\xi})$. Then $\mathbf{N}\langle \mathrm{G} \rangle^{\bot} <: \mathbf{N}\langle \mathrm{G}, \zeta \rangle <: \mathbf{N}\langle \mathrm{G} \rangle$ and $\mathbf{E}\langle \mathrm{G} \rangle^{\bot} <: \mathbf{E}\langle \mathrm{G}, \xi \rangle <: \mathbf{E}\langle \mathrm{G} \rangle$ for $\zeta \in \overline{\zeta}$ and $\xi \in \overline{\xi}$, respectively.

Definition 3.3 (Record and Binding Table Schemas). The schema $\Gamma : \mathcal{A} \rightharpoonup \tau$ of a record specifies the type for each binding variable in its domain. The schema of a binding table is the union (Definition 3.4) of the schemas of all the records in its support.

Definition 3.4 (Record Schema Unions and Joins). Let Γ_1 and Γ_2 be two record schemas.

$[\Gamma_1 \sim_{\cup} \Gamma_2]$. They are *union compatible* when $\mathbf{dom}(\Gamma_1) = \mathbf{dom}(\Gamma_2)$.

$[\Gamma_1 \sim_{\bowtie} \Gamma_2]$. They are *join compatible* when $\forall x \in \mathbf{dom}(\Gamma_1) \cap \mathbf{dom}(\Gamma_2)$:

- $\Gamma_1(x) <: \mathbf{N}\langle - \rangle$ (or $\mathbf{E}\langle - \rangle$) and $\Gamma_2(x) <: \mathbf{N}\langle - \rangle$ (or $\mathbf{E}\langle - \rangle$); and
- $\Gamma_1(x) \cap \Gamma_2(x) \neq \bot$.

$[\Gamma_1 \cup \Gamma_2]$. If $\Gamma_1 \sim_{\cup} \Gamma_2$, their *union* is: $\Gamma_1 \cup \Gamma_2(x) \triangleq \Gamma_1(x) \cup \Gamma_2(x), x \in \mathbf{dom}(\Gamma_1)$.

$[\Gamma_1 \bowtie \Gamma_2]$. If $\Gamma_1 \sim_{\bowtie} \Gamma_2$, their *join* is:

$$\Gamma_1 \bowtie \Gamma_2(x) \triangleq \begin{cases} \Gamma_1(x) & x \in \mathbf{dom}(\Gamma_1) \setminus \mathbf{dom}(\Gamma_2) \\ \Gamma_2(x) & x \in \mathbf{dom}(\Gamma_2) \setminus \mathbf{dom}(\Gamma_1) \\ \Gamma_1(x) \cap \Gamma_2(x) & x \in \mathbf{dom}(\Gamma_1) \cap \mathbf{dom}(\Gamma_2) \end{cases}$$

4 Well-formedness of GQL Queries

We now use our type system (§3) to define well-formedness rules for GQL queries and assign binding table schemas to their results. GQL's native support for graph schemas (ISO §4.13) allows fine-grained static typing. Schema information flows from typing rules for pattern atoms through patterns and queries, providing static guarantees that are unavailable in schema-free formalisms.

We organize GQL query well-formedness into three layers that mirror the syntactic hierarchy of our calculus in Figure 2: value expressions, patterns (§4.2), and queries (§4.3). This organization simplifies the type-soundness argument (§6) by allowing it to proceed compositionally, lifting guarantees from expressions to patterns and ultimately to queries. We only consider queries where all pattern atoms are named (non-anonymous), i.e., they declare their binding variables: $\mathbf{var}(A) \neq \epsilon$.

4.1 GQL Value Expressions

Due to space constraints, we only introduce the typing judgement for GQL value expressions here for clarity when referencing them in later sections. We type value expressions under the judgement: $\Sigma; \mathrm{G}; \Gamma \vdash^{\square}_{\diamond} \vartheta : \tau \triangleright \mathcal{X}$, which assigns type τ to the value expression ϑ under the context containing: the database world (Σ), a working graph site G, a record schema (Γ), the binding variables used in the expression ($\mathcal{X}$), the permitted number of aggregate functions ($\square \in \{0, 1\}$), and the reference-mode ($\diamond \in \{\mathbb{1}, \bigstar\}$) that captures if the expression is inside or outside a quantifier context.

4.2 GQL Patterns

Typing patterns is one of the most technically challenging aspects of defining well-formedness of GQL queries because they introduce all bindings to graph elements, so their typing must statically

predict the schema that the operational semantics produces at runtime. Moreover, since our GQL formalism includes graphs with schemas, typing becomes *strictly more precise* than in schema-free approaches. For instance, a binding variable can be statically assigned $\tau^{\perp}$ when the schema's label constraints together with the pattern rule out any valid homomorphism. These refinements compose in non-trivial ways because: (a) *surrounding context*—a pattern list is composed of several path patterns that may have overlaps in their declared binding variables, and (b) *internal context*—a pattern is composed of several pattern atoms that may similarly have overlaps in their declared binding variables. We reconcile the two by structuring our typing rules across different levels that mirror the grammar of Figure 2: (1) *atom typing* types individual node and edge atoms in isolation from all context; (2) *path pattern typing* composes atoms only under internal context, staying isolated from surrounding context; and (3) *pattern list typing* composes path patterns via conjunction under surrounding context. Levels (2) and (3) refine overlapping variables left-to-right in one forward pass without fixed-point iteration, which may assign wider types than necessary. We formalize pattern well-formedness using three main judgement forms.

$$\Sigma; \mathsf{G} \vdash_{\mathsf{Atom}} A : \Gamma \qquad \Sigma; \mathsf{G} \vdash^{\diamond}_{\mathsf{Pat}} P : \Gamma \qquad \Sigma; \mathsf{G} \vdash_{\mathsf{PatLst}} p : \Gamma$$

The *atom judgement* types a pattern atom A, i.e., a node N or an edge E atom, in isolation to produce a singleton binding table schema Γ. The *pattern judgement* composes atoms into path patterns. Its reference mode $\diamond \in \{\mathbb{1}, \bigstar\}$ records whether the pattern sits outside ($\mathbb{1}$) or inside ($\bigstar$) a quantifier context. The *pattern list judgement* composes path patterns using conjunction (p, P).

Atom Typing (Level 1). Atom typing resolves the label expression l and property map Π of a pattern atom A against the graph schema $(\overline{\zeta}, \overline{\xi})$, producing a schema-refined type. Two auxiliary judgements perform the filtering and operate uniformly on node and edge schemas, so we write ς for a schema entry of either kind. The judgement $\overline{\varsigma} \vdash_{\mathsf{Lbl}} \mathbf{lbl}(A) \rightsquigarrow \overline{\varsigma}'$ filters element schemas by structural induction on the label expression, covering: labels L, the wildcard ($\%$), conjunction ($\&$), disjunction ($|$), and negation ($!$). The judgement $\overline{\varsigma} \vdash_{\mathsf{Prp}} \mathbf{prp}(A) \rightsquigarrow \overline{\varsigma}'$ similarly filters element schemas, but by property maps. Intersecting their results yields the compatible element schemas.

$$\textsc{Sch-Lbl-Empty}\ \frac{}{\overline{\varsigma} \vdash_{\mathsf{Lbl}} \varepsilon \rightsquigarrow \overline{\varsigma}} \qquad \textsc{Sch-Lbl-Atom}\ \frac{\overline{\varsigma}_L := \{\varsigma \in \overline{\varsigma} \mid L \in \boldsymbol{\pi_1}(\varsigma)\}}{\overline{\varsigma} \vdash_{\mathsf{Lbl}} L \rightsquigarrow \overline{\varsigma}_L}$$

$$\textsc{Sch-Lbl-Negation}\ \frac{\overline{\varsigma} \vdash_{\mathsf{Lbl}} l \rightsquigarrow \overline{\varsigma}_l}{\overline{\varsigma} \vdash_{\mathsf{Lbl}} !l \rightsquigarrow \overline{\varsigma} \setminus \overline{\varsigma}_l} \qquad \textsc{Sch-Lbl-Conjunction}\ \frac{\overline{\varsigma} \vdash_{\mathsf{Lbl}} l_1 \rightsquigarrow \overline{\varsigma}_1 \qquad \overline{\varsigma} \vdash_{\mathsf{Lbl}} l_2 \rightsquigarrow \overline{\varsigma}_2}{\overline{\varsigma} \vdash_{\mathsf{Lbl}} l_1 \,\&\, l_2 \rightsquigarrow \overline{\varsigma}_1 \cap \overline{\varsigma}_2}$$

$$\textsc{Sch-Lbl-Disjunction}\ \frac{\overline{\varsigma} \vdash_{\mathsf{Lbl}} l_1 \rightsquigarrow \overline{\varsigma}_1 \qquad \overline{\varsigma} \vdash_{\mathsf{Lbl}} l_2 \rightsquigarrow \overline{\varsigma}_2}{\overline{\varsigma} \vdash_{\mathsf{Lbl}} l_1 \mid l_2 \rightsquigarrow \overline{\varsigma}_1 \cup \overline{\varsigma}_2} \qquad \textsc{Sch-Lbl-Wildcard}\ \frac{\overline{\varsigma}_{\%} := \{\varsigma \in \overline{\varsigma} \mid \boldsymbol{\pi_1}(\varsigma) \neq \varnothing\}}{\overline{\varsigma} \vdash_{\mathsf{Lbl}} \% \rightsquigarrow \overline{\varsigma}_{\%}}$$

$$\textsc{Sch-Prp-Atom}\ \frac{\overline{\varsigma}_{\Phi} := \{\varsigma \in \overline{\varsigma} \mid \Phi \subseteq \boldsymbol{\pi_2}(\varsigma)\}}{\overline{\varsigma} \vdash_{\mathsf{Prp}} \Phi \rightsquigarrow \overline{\varsigma}_{\Phi}}$$

Sch-Lbl-Empty returns the schema set as is since there is no label expression. Sch-Lbl-Atom retains schemas whose label set contains L, while Sch-Lbl-Wildcard retains schemas with at least one label. Sch-Lbl-Negation, Sch-Lbl-Conjunction, and Sch-Lbl-Disjunction filter schemas based on logical operations corresponding to their names. Sch-Prp-Atom filters by property maps.

$$\textsc{Prp-Empty}\ \frac{}{\vdash_{\mathsf{Prp}} \varnothing : \varnothing} \qquad \textsc{Prp-Atom}\ \frac{k \in \mathcal{A} \qquad \tau \in \mathcal{T}_0 \qquad c \in \mathscr{U}_\tau}{\vdash_{\mathsf{Prp}} [k \mapsto c] : [k \mapsto \tau]}$$

$$\textsc{Prp-Insert}\ \frac{k \notin \mathbf{dom}(\Pi) \qquad \vdash_{\mathsf{Prp}} \Pi : \Phi_\Pi \qquad \vdash_{\mathsf{Prp}} [k \mapsto c] : \Phi_k}{\vdash_{\mathsf{Prp}} \Pi \sqcup [k \mapsto c] : \Phi_\Pi \sqcup \Phi_k}$$

The judgement $\vdash_{\mathsf{Prp}} \Pi : \Phi$ types a property map Π with a property schema Φ. Prp-Empty types the empty map, Prp-Atom types a singleton map by inferring the constant's scalar type, and Prp-Insert combines property schemas via disjoint union provided their domains are also disjoint.

$$\textsc{Sch-Atom}\quad \frac{\mathbf{var}(A) \in \mathcal{A} \qquad \vdash_{\mathsf{Prp}} \mathbf{prp}(A) : \Phi \qquad \overline{\varsigma} \vdash_{\mathsf{Lbl}} \mathbf{lbl}(A) \rightsquigarrow \overline{\varsigma}_{\mathsf{l}} \qquad \overline{\varsigma} \vdash_{\mathsf{Prp}} \Phi \rightsquigarrow \overline{\varsigma}_{\Phi}}{\overline{\varsigma} \vdash_{\mathsf{Atom}} A \rightsquigarrow \overline{\varsigma}_{\mathsf{l}} \cap \overline{\varsigma}_{\Phi}}$$

$$\textsc{Atom-Node-Open}\quad \frac{\mathrm{G} \notin \mathbf{dom}(\Upsilon) \qquad \mathbf{var}(N) \in \mathcal{A}}{\Sigma; \mathrm{G} \vdash_{\mathsf{Atom}} N : [\mathbf{var}(N) \mapsto \mathbf{N}\langle \mathrm{G} \rangle]}$$

$$\textsc{Atom-Node-Closed}\quad \frac{\mathrm{G} \in \mathbf{dom}(\Upsilon) \qquad (\overline{\zeta}, -) = \Upsilon(\mathrm{G}) \qquad \overline{\zeta} \vdash_{\mathsf{Atom}} N \rightsquigarrow \overline{\zeta}_N}{\Sigma; \mathrm{G} \vdash_{\mathsf{Atom}} N : [\mathbf{var}(N) \mapsto \mathbf{N}\langle \mathrm{G}, \overline{\zeta}_N \rangle]}$$

$$\textsc{Atom-Edge-Open}\quad \frac{\mathrm{G} \notin \mathbf{dom}(\Upsilon) \qquad \mathbf{var}(E) \in \mathcal{A}}{\Sigma; \mathrm{G} \vdash_{\mathsf{Atom}} E : [\mathbf{var}(E) \mapsto \mathbf{E}\langle \mathrm{G} \rangle]}$$

$$\textsc{Atom-Edge-Closed}\quad \frac{\mathrm{G} \in \mathbf{dom}(\Upsilon) \qquad (-, \overline{\xi}) = \Upsilon(\mathrm{G}) \qquad \overline{\xi} \vdash_{\mathsf{Atom}} E \rightsquigarrow \overline{\xi}_E}{\Sigma; \mathrm{G} \vdash_{\mathsf{Atom}} E : [\mathbf{var}(E) \mapsto \mathbf{E}\langle \mathrm{G}, \overline{\xi}_E \rangle]}$$

Sch-Atom resolves A's schema by intersecting its label-expression and property map filter results. The atom-typing rules handle open and closed graphs separately. Open graphs (Atom-Node-Open) have no schema, so the atom's binding variable $\mathbf{var}(N)$ is assigned just the graph-indexed type $\mathbf{N}\langle \mathrm{G} \rangle$. For closed graphs (Atom-Node-Closed), the variable is assigned the schema-refined type $\mathbf{N}\langle \mathrm{G}, \overline{\zeta}_N \rangle$. If the atom's label-expression and property map combination does not satisfy the constraints of any node schema in G's graph schema, i.e., $\overline{\zeta}_N = \varnothing$; the variable is assigned the annotated empty type $\mathbf{N}\langle \mathrm{G} \rangle^{\perp}$ since $\mathbf{N}\langle \mathrm{G}, \varnothing \rangle \triangleq \mathbf{N}\langle \mathrm{G} \rangle^{\perp}$ (§ 3.4). Symmetric rules cover edge atoms.

Path Pattern Typing (Level 2). Path pattern typing composes atoms into complete path patterns. When composing a node–edge–node step $N_1 E_2 N_2$, the types assigned to the atoms—N_1, E_2, and N_3—in isolation may be overly permissive. This is because *atom typing* (Level 1) types atoms with refined element schemas, with refinement based solely on their label-expressions and property maps. While this "may be" sufficient for typing the node atoms N_1 and N_2, it is not the case for the edge atom E_2, as edge schemas impose additional constraints beyond label sets and property maps: endpoint node schemas and edge directionality (Definition 2.2). Moreover, refining E_2's type based on the additional constraints usually requires refining the types of its endpoints N_1 and N_2 as well.

We formalize path pattern typing such that type refinement for the composing node-edge-node step is done jointly, allowing typing to prune infeasible triples and thus assign more precise types. For the path pattern $P\,E\,N$, the judgement $\Sigma; \mathrm{G}; \Gamma_P; \Gamma_E; \Gamma_N \vdash_{\mathsf{Rfn}} P\,E\,N \rightsquigarrow \Gamma_P', \Gamma_E', \Gamma_N'$ jointly refines the types of the binding variables corresponding to the rightmost node atom (i.e., $\mathbf{tail}(\cdot)$) of the preceding pattern P, the edge atom E, and the following node atom N; from their individual table schemas (typed in isolation)—Γ_P, Γ_E, and Γ_N—to produce the type refined schemas Γ_P', Γ_E', and Γ_N'.

Definition 4.1 (Type Preserving Update for Schema-Refined Types). Consider a graph site G with graph schema $(\overline{\zeta}, \overline{\xi})$. Let τ be a schema-refined type $\mathbf{N}\langle \mathrm{G}, - \rangle$ (or $\mathbf{E}\langle \mathrm{G}, - \rangle$) or its list-lifted version; and let $\overline{\varsigma}, \overline{\varsigma}'$ range over the subsets of $\overline{\zeta}$ (or $\overline{\xi}$). We then define the type preserving update $\tau\restriction_{\overline{\varsigma}}$ as:

$$\tau\restriction_{\overline{\varsigma}} \triangleq \begin{cases} \mathbf{List}\,(\lfloor \tau \rfloor[\overline{\varsigma}/\overline{\varsigma}']) & \text{if } \tau = \mathbf{List}\,\mathbf{N}\langle \mathrm{G}, \overline{\varsigma}' \rangle \textbf{ or } \mathbf{List}\,\mathbf{E}\langle \mathrm{G}, \overline{\varsigma}' \rangle \\ \tau[\overline{\varsigma}/\overline{\varsigma}'] & \text{if } \tau = \mathbf{N}\langle \mathrm{G}, \overline{\varsigma}' \rangle \textbf{ or } \mathbf{E}\langle \mathrm{G}, \overline{\varsigma}' \rangle \end{cases}$$

The operation $\tau[\overline{\varsigma}/\overline{\varsigma}']$ denotes replacing the schema set $\overline{\varsigma}'$ of the schema-refined type τ with the set $\overline{\varsigma}$. It yields the appropriate annotated empty type when $\overline{\varsigma} = \varnothing$ (e.g., $\mathbf{N}\langle \mathrm{G}, \overline{\zeta}' \rangle[\varnothing/\overline{\zeta}'] = \mathbf{N}\langle \mathrm{G} \rangle^{\perp}$).

$$\textsc{Refine-Open}\quad \frac{\mathrm{G} \notin \mathbf{dom}(\Upsilon) \qquad \sim_{\bowtie}(\Gamma_P, \Gamma_E, \Gamma_N)}{\Sigma; \mathrm{G}; \Gamma_P; \Gamma_E; \Gamma_N \vdash_{\mathsf{Rfn}} P\,E\,N \rightsquigarrow \Gamma_P, \Gamma_E, \Gamma_N} \qquad \boldsymbol{\theta}_E(\zeta_1, \xi_2, \zeta_3) \triangleq \bigvee_{\partial \in \mathbf{ornt}(E)} \begin{cases} (\zeta_1, \zeta_3, \circlearrowright) = (\boldsymbol{\pi}_3(\xi_2), \boldsymbol{\pi}_4(\xi_2), \boldsymbol{\pi}_5(\xi_2)) & \text{if } \partial = \rightarrow \\ (\zeta_1, \zeta_3, \circlearrowright) = (\boldsymbol{\pi}_4(\xi_2), \boldsymbol{\pi}_3(\xi_2), \boldsymbol{\pi}_5(\xi_2)) & \text{if } \partial = \leftarrow \\ (\zeta_1, \zeta_3, \bowtie) = (\boldsymbol{\pi}_3(\xi_2), \boldsymbol{\pi}_4(\xi_2), \boldsymbol{\pi}_5(\xi_2)) & \text{else} \\ \quad\textbf{or} & \\ (\zeta_1, \zeta_3, \bowtie) = (\boldsymbol{\pi}_4(\xi_2), \boldsymbol{\pi}_3(\xi_2), \boldsymbol{\pi}_5(\xi_2)) & \end{cases}$$

$$\textsc{Refine-Closed}\quad \frac{\begin{array}{c}\mathrm{G} \in \mathbf{dom}(\Upsilon) \qquad \sim_{\bowtie}(\Gamma_P, \Gamma_E, \Gamma_N) \qquad (\tau_1, \tau_2, \tau_3) := (\Gamma_P(\mathbf{tail}(P)), \Gamma_E(\mathbf{var}(E)), \Gamma_N(\mathbf{var}(N))) \\ \lfloor \tau_1 \rfloor \simeq_? \mathbf{N}\langle \mathrm{G}, \overline{\zeta}_1 \rangle \quad \lfloor \tau_2 \rfloor \simeq_? \mathbf{E}\langle \mathrm{G}, \overline{\xi}_2 \rangle \quad \lfloor \tau_3 \rfloor \simeq_? \mathbf{N}\langle \mathrm{G}, \overline{\zeta}_3 \rangle \qquad \mathcal{S} := \{(\zeta_1, \xi_2, \zeta_3) \in (\overline{\zeta}_1 \times \overline{\xi}_2 \times \overline{\zeta}_3) \mid \boldsymbol{\theta}_E(\zeta_1, \xi_2, \zeta_3)\}\end{array}}{\Sigma; \mathrm{G}; \Gamma_P; \Gamma_E; \Gamma_N \vdash_{\mathsf{Rfn}} P\,E\,N \rightsquigarrow \Gamma_P\big[\mathbf{tail}(P) \mapsto \tau_1 \restriction_{\boldsymbol{\pi}_1(\mathcal{S})}\big],\ \Gamma_E\big[\mathbf{var}(E) \mapsto \tau_2 \restriction_{\boldsymbol{\pi}_2(\mathcal{S})}\big],\ \Gamma_N\big[\mathbf{var}(N) \mapsto \tau_3 \restriction_{\boldsymbol{\pi}_3(\mathcal{S})}\big]}$$

Refine-Open handles type refinement in open graphs and simply checks pairwise join compatibility (Definition 3.4, via shorthand: $\sim_{\bowtie}(\ldots)$) of the individual schemas corresponding to the preceding pattern P, edge atom E, and following node atom N; before returning them unchanged. Refine-Closed handles closed graphs. Let τ_1, τ_2, and τ_3 correspond to the types assigned in isolation to the binding variables: $\mathbf{tail}(P)$, $\mathbf{var}(E)$, and $\mathbf{var}(N)$, respectively. We first check if the types are the appropriate schema-refined types up to nullability, using the shorthand: $\tau' \simeq_? \tau \triangleq \tau <: \tau' <: \tau?$. Next, we form the Cartesian product of their schema sets, retaining only the triples $(\zeta_1, \xi_2, \zeta_3)$ that satisfy the *additional constraints* imposed by edge schemas (ξ_2) via the endpoint condition $\boldsymbol{\theta}_E(\zeta_1, \xi_2, \zeta_3)$. The endpoint condition holds when the node-edge-node schema triple is compatible under some orientation in $\mathbf{ornt}(E)$ (Figure 4). Each orientation ($\partial \in \{\rightarrow, \leftarrow, \sim\}$) compares the endpoint node schemas ζ_1 and ζ_3 against that of the edge schema ξ_2 as a tuple, with ordering determined by the orientation; the edge directionality $\boldsymbol{\pi}_5(\xi_2)$ must match $\mathbf{dir}(\partial)$. Orientation "$\rightarrow$" takes the endpoint node schemas in order, "$\leftarrow$" swaps them, but both require the edge to be directed; while "$\sim$" compares them as an unordered pair for undirected edges. Finally, the filtered schema triples are used to update the variables' types via the type-preserving update from Definition 4.1.

Definition 4.2 (Quantifier Lift). Given a record schema Γ and a quantifier K, the quantifier lift operation $\Gamma\uparrow_K$ is defined as: $\forall x \in \mathbf{dom}(\Gamma)$:

$$\Gamma\uparrow_K (x) \triangleq \begin{cases} \Gamma(x)? & \text{if } K = ? \\ \mathbf{List}\ \Gamma(x) & \text{otherwise} \end{cases}$$

Path patterns are described in ISO §16.7, and SR:8 (Page 229) states that patterns may only be quantified if they have a minimum path length of one. We enforce this during path pattern typing via the premise $\mathbf{lo}^{\#\mathbf{E}}(P) > 0$. SR:16 (Page 231) imposes another restriction of path patterns having at least one node atom regardless of quantification. We account for this while typing pattern lists.

$$\textsc{Pat-Node}\quad \frac{\Sigma; \mathrm{G} \vdash_{\mathsf{Atom}} N : \Gamma}{\Sigma; \mathrm{G} \vdash^{\diamond}_{\mathsf{Pat}} N : \Gamma} \qquad \textsc{Pat-Step}\quad \frac{\Sigma; \mathrm{G} \vdash^{\diamond}_{\mathsf{Pat}} P : \Gamma_P \qquad \Sigma; \mathrm{G} \vdash^{\diamond}_{\mathsf{Pat}} N : \Gamma_N \qquad \Sigma; \mathrm{G}; \Gamma_P; \Gamma_E; \Gamma_N \vdash_{\mathsf{Rfn}} P\,E\,N \rightsquigarrow \Gamma_P', \Gamma_E', \Gamma_N'}{\Sigma; \mathrm{G}; \Gamma_E \vdash^{\diamond}_{\mathsf{PatStep}} P\,E\,N : \Gamma_P' \bowtie \Gamma_E' \bowtie \Gamma_N'}$$

$$\textsc{Pat-Edge}\quad \frac{\Sigma; \mathrm{G} \vdash_{\mathsf{Atom}} E : \Gamma_E \qquad \Sigma; \mathrm{G}; \Gamma_E \vdash^{\diamond}_{\mathsf{PatStep}} P\,E\,N : \Gamma}{\Sigma; \mathrm{G} \vdash^{\diamond}_{\mathsf{Pat}} P\,E\,N : \Gamma} \qquad \textsc{Pat-Quant-Edge}\quad \frac{\Sigma; \mathrm{G} \vdash_{\mathsf{Atom}} E : \Gamma_E \qquad \Sigma; \mathrm{G}; (\Gamma_E\uparrow_K) \vdash^{\star}_{\mathsf{PatStep}} P\,E\,N : \Gamma}{\Sigma; \mathrm{G} \vdash^{\mathbb{I}}_{\mathsf{Pat}} P\,E\,K\,N : \Gamma}$$

$$\textsc{Pat-Paren-Path}\quad \frac{\Sigma; \mathrm{G} \vdash^{\diamond}_{\mathsf{Pat}} P : \Gamma}{\Sigma; \mathrm{G} \vdash^{\diamond}_{\mathsf{Pat}} (P) : \Gamma} \qquad \textsc{Pat-Quant-Path}\quad \frac{\mathbf{lo}^{\#\mathbf{E}}(P) > 0 \qquad \Sigma; \mathrm{G} \vdash^{\star}_{\mathsf{Pat}} (P) : \Gamma}{\Sigma; \mathrm{G} \vdash^{\mathbb{I}}_{\mathsf{Pat}} (P) K : \Gamma\uparrow_K}$$

Pat-Node lifts a node atom as a path pattern. Pat-Step is an auxiliary judgement that type refines the binding variables of a path pattern using the refinement rules described earlier. Pat-Edge

introduces an edge atom, delegating to Pat-Step. Pat-Quant-Edge handles quantified edges by first lifting their singleton table schemas through $(\cdot)\uparrow_K$ from Definition 4.2 before delegating to Pat-Step (similar to Pat-Edge). Pat-Paren-Path types parenthesized path patterns by simply typing the enclosed pattern. Pat-Quant-Path handles quantified path patterns by checking if they have at least one edge atom, and then typing them, before lifting their table schemas through $(\cdot)\uparrow_K$.

Pattern List Typing (Level 3). Pattern list typing composes path patterns via conjunction (p, P). We account for the minimum node count restriction imposed by SR:16 (Page 231) here, via $\mathbf{lo}^{\#\mathbf{N}}(P) > 0$.

$$\textsc{PatList-Lift}\ \frac{\mathbf{lo}^{\#\mathbf{N}}(P) > 0 \qquad \Sigma; \mathrm{G} \vdash^{\mathbb{1}}_{\mathsf{Pat}} P : \Gamma}{\Sigma; \mathrm{G} \vdash_{\mathsf{PatLst}} P : \Gamma} \qquad \textsc{PatList-Conjunction}\ \frac{\Sigma; \mathrm{G} \vdash_{\mathsf{PatLst}} p : \Gamma_p \qquad \Sigma; \mathrm{G} \vdash_{\mathsf{PatLst}} P : \Gamma_P \qquad \Gamma_p \sim_{\bowtie} \Gamma_P}{\Sigma; \mathrm{G} \vdash_{\mathsf{PatLst}} p, P : \Gamma_p \bowtie \Gamma_P}$$

Pat-List-Lift lifts a path pattern into a pattern list if it has at least one node atom. Pat-List-Conjunction types a conjunction. It types both sub-components—the preceding path pattern list p and the following path pattern P—independently, and verifies the join compatibility of their table schemas Γ_p and Γ_P before joining them to produce a single table schema for the conjunction. Since path pattern lists are typed under surrounding context in one forward left-to-right pass, only those variables declared in p that overlap with the ones in P are refined during the schema join $\Gamma_p \bowtie \Gamma_P$.

4.3 GQL Queries

GQL queries are either focused linear queries (Q) that match patterns against a single working graph and optionally filter the matched bindings before projecting them via projection expressions; or composite queries (q) that compose several focused linear queries via composite operators ($\circledast$).

Query typing is relatively simple since much of the complexity has been deliberately offloaded to typing: (1) value expressions (§4.1)—used for typing the predicate and projection expressions in **Where** and **Return** clauses, respectively; and (2) patterns (§4.2)—used for typing the patterns in **Match** clauses. We type GQL queries under three judgement forms,

$$\underset{\text{Projection}}{\Sigma; \mathrm{G}; \Gamma \vdash \mu : \Gamma_\mu} \qquad \underset{\text{Query}}{\Sigma \vdash Q : \Gamma} \qquad \underset{\text{Composite Query}}{\Sigma \vdash_{\circledast} q : \Gamma}$$

The *projection judgement* types projection expressions by projecting an upstream table schema in the context of a working graph. The projected table schema is the schema of a focused linear query's result binding table. The *query judgement* assigns focused linear queries with the schemas of their result binding tables. The *composite query judgement* types composite queries similarly, but under the additional context of the composite operators used for focused linear query composition.

Projection Typing. Projection expressions project upstream table schemas (Γ) using value expressions (ϑ). Projection expressions are covered in ISO §14.11, and its SR:8 (Page 186) states that they must be aliased via **As** unless they are just references to binding variables ($x \in \mathbf{dom}(\Gamma)$). We choose the fragment for projection expressions and structure its typing rules to match the standard.

$$\textsc{Prj-Atom-ValExp}\ \frac{x \in \mathcal{A} \qquad \Sigma; \mathrm{G}; \Gamma \vdash^{1}_{\bigstar} \vartheta : \tau \triangleright \mathcal{X}}{\Sigma; \mathrm{G}; \Gamma \vdash \vartheta\ \mathbf{As}\ x : [x \mapsto \tau]} \qquad \textsc{Prj-Atom-VarRef}\ \frac{}{\Sigma; \mathrm{G}; \Gamma \vdash x\ \mathbf{As}\ x : \Gamma_x} \qquad \textsc{Prj-Compose}\ \frac{\Sigma; \mathrm{G}; \Gamma \vdash \mu : \Gamma_\mu \qquad \Sigma; \mathrm{G}; \Gamma \vdash \_\mu : \Gamma_{\_\mu} \qquad \mathbf{dom}(\Gamma_\mu) \cap \mathbf{dom}(\Gamma_{\_\mu}) = \varnothing}{\Sigma; \mathrm{G}; \Gamma \vdash \mu, \_\mu : \Gamma_\mu \bowtie \Gamma_{\_\mu}}$$

Prj-Atom-ValExp covers value expressions generally, and so compulsorily includes an alias. It types value expressions in the context of an upstream table schema, and produces a singleton table schema with the alias mapped to that type. Value expressions that are just binding variable

references are handled by Prj-Atom-VarRef, which first rewrites them vacuously into aliased projection expressions, before delegating them to Prj-Atom-ValExp. Prj-Compose covers projection composition by typing their atoms left-to-right, joining their schemas only when they are disjoint.

Linear Query Typing. Linear queries first select a working graph via **Use**, then introduce binding tables via **Match** and optionally filter them via **Where**, before projecting them using **Return**.

$$\textsc{Qry-Match-Filter}\quad \frac{\mathsf{G} \in \mathbf{dom}(\Omega) \qquad \Sigma;\mathsf{G} \vdash_{\mathsf{PatLst}} p : \Gamma_p \qquad \Sigma;\mathsf{G};\Gamma_p \vdash^{1}_{\mathbb{I}} \phi : \mathbb{B}? \triangleright \mathcal{X} \qquad \Sigma;\mathsf{G};\Gamma_p \vdash \mu : \Gamma}{\Sigma \vdash \textbf{Use}\ \mathsf{G}\ \textbf{Match}\ p\ \textbf{Where}\ \phi\ \textbf{Return}\ \mu : \Gamma}$$

$$\textsc{Qry-Match}\quad \frac{\Sigma \vdash \textbf{Use}\ \mathsf{G}\ \textbf{Match}\ p\ \textbf{Where}\ \textbf{True}\ \textbf{Return}\ \mu : \Gamma}{\Sigma \vdash \textbf{Use}\ \mathsf{G}\ \textbf{Match}\ p\ \textbf{Return}\ \mu : \Gamma}$$

Qry-Match-Filter checks if the specified graph G is in the catalog Ω, and types the pattern expression p of the **Match** clause to obtain its table schema Γ_p, which is used for typing the predicate of the **Where** clause as a nullable Boolean in singleton-reference context; and the projection expression of the **Return** clause to obtain the result table schema Γ. Qry-Match first rewrites with a tautological predicate **True** for **Where**, and then delegates to Qry-Match-Filter.

Composite Query Typing. Composite queries compose linear queries under composite operators such as **Union** and **Except**, etc., with an optional qualifier (**Distinct** v. **All**) for choosing set or bag semantics for the operation. Composite queries are covered in ISO §14.2, and its SR:3 (Page 164) states that the composite operators used within a single composite query must all be identical (including the qualifier that decides set v. bag semantics). We design our typing rules to match this.

$$\textsc{CompQry-Lift}\quad \frac{\Sigma \vdash Q : \Gamma}{\Sigma \vdash_{\circledast} Q : \Gamma}$$

$$\textsc{CompQry-Compose}\quad \frac{\Sigma \vdash_{\circledast} q : \Gamma_q \qquad \Sigma \vdash_{\circledast} Q : \Gamma_Q \qquad \Gamma_q \sim_{\circledast} \Gamma_Q}{\Sigma \vdash_{\circledast} q \circledast Q : \Gamma_q \circledast \Gamma_Q}$$

CompQry-Lift lifts a linear query into the composite judgement. CompQry-Compose covers query composition by typing the linear query atoms in the context of the composite operator $\circledast$, checking compatibility, and combining their table schemas based on the operator's semantics. Since the judgement carries the composite operator in its context, it automatically enforces the GQL standard's requirement of a single composite query using the same composite operator throughout.

5 Small-Step Operational Semantics

We defined well-formedness of GQL queries in §4 by structuring our typing rules compositionally, i.e., lifting context from query-level down to pattern- and expression-level, and typing them with binding table schemas. We now formally define how these queries are executed. While most prior works have formalized query language semantics denotationally in general, including GQL [11], we use a small-step operational approach. This allows us to explicitly show the complexities of interleaving states during query execution, arising from the non-trivial semantics of GQL.

Since GQL allows queries to specify multiple ways of traversing graphs for pattern matching (ISO §4.11), we first fix the variant that we formalize. Graph pattern matching semantics is configured primarily, and jointly, through two parameters: *path modes* (ISO §4.11.7) and *match modes* (ISO §4.11.9). The path modes impose constraints on matched edge/node frequencies for individual path patterns (P), while the match modes control whether these constraints are enforced across the entire pattern list (p). Since every successful pattern match implies a new record containing bindings to the matched elements, the action of path and match modes can be understood as filtering the binding tables of the composing path patterns and pattern lists, respectively. Path modes filter

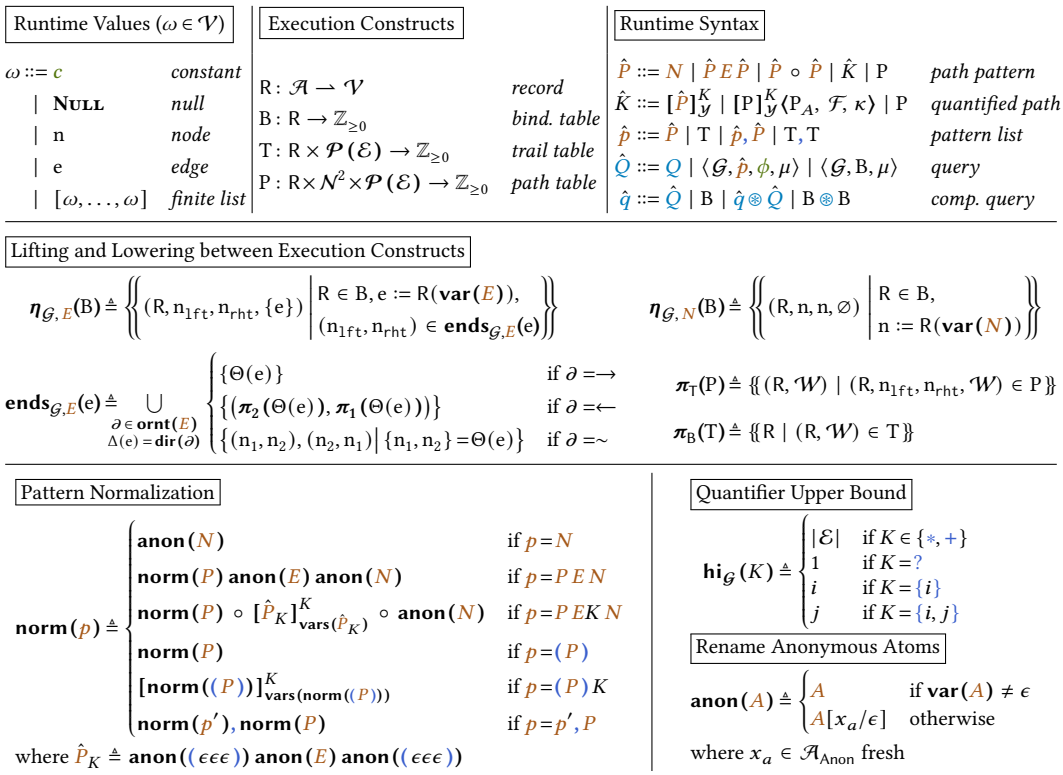


Fig. 5. Runtime values, execution constructs, runtime syntax, and runtime metafunctions used in our semantics formalization. Runtime terms are shown with a hat (e.g., $\hat{P}$) to differentiate them from the source terms of Figure 2. P_A and $\mathcal{F}$ are the accumulator and the frontier of a quantified path, and κ counts its repetitions.

a path pattern's records, which always represent contiguous paths in a graph, by: (1) `TRAIL`—graph edges may not be bound more than once; (2) `ACYCLIC`—graph nodes may not be bound more than once, i.e., matched paths must be acyclic; (3) `SIMPLE`—similar to `ACYCLIC` except, graph nodes bound to the head/tail variables of a path pattern may repeat, i.e., matched paths may be cyclic; and (4) `WALK`—no constraints. Match modes filter a pattern list's records by: (1) `DIFFERENT EDGES`—graph edges may not be bound more than once; and (2) `REPEATABLE ELEMENTS`—no additional constraints.

We use the `TRAIL` path mode and `DIFFERENT EDGES` match mode in our formalization, as they are the default in popular graph query languages like Cypher, and because they guarantee termination of pattern matching, i.e., graph traversal during pattern matching terminates (regardless of quantifiers) since traversed paths cannot repeat edges and hence are upper-bounded by the graph's edge count.

We start by introducing the runtime values and syntax used in our formalization (§5.1); before moving onto the actual GQL semantics, which similar to our typing rules, is organized into three layers mirroring the calculus in Figure 2: (1) *Value Expressions* (not shown due to space constraints); (2) *Patterns* (§5.2)—graph pattern matching; and (3) *Queries* (§5.3)—the full query evaluation pipeline.

5.1 Runtime Values, Execution Constructs, and Runtime Syntax

Figure 5 summarizes the runtime values, execution constructs, runtime syntax, and metafunctions used in our semantics formalization. The *runtime value domain* $\mathcal{V}$ extends the multi-sorted value

universe $\mathscr{U}$ from §2.1 with property graph elements—nodes (n) and edges (e)—and finite lists. The former ranges over the node/edge sets of all the graphs in a database instance's catalog Ω, while the latter is constructed at runtime for accumulating bindings across quantified pattern iterations. Lists are also paired with the standard *list concatenation* operation "::" for concatenating two lists.

Definition 5.1 (Records and Binding Tables). A *record* $\mathsf{R} : \mathcal{A} \rightharpoonup \mathcal{V}$, is a partial function from binding variables to runtime values. Two records are *compatible*: $\mathsf{R}_1 \asymp \mathsf{R}_2$, when they agree on their common domain; their natural join: $\mathsf{R}_1 \bowtie \mathsf{R}_2$ is then their union as partial functions, and $\bot$ otherwise. A *binding table* $\mathsf{B} : \mathsf{R} \to \mathbb{Z}_{\geq 0}$, is a bag of records R; their join is the join of all compatible record pairs.

The GQL standard specifies two *execution constructs*: records (R) and binding tables (B), which are used throughout the entire query evaluation pipeline to hold intermediate as well as the final results (Definition 5.1). However, since we consider the `TRAIL` path mode under the `DIFFERENT EDGES` matching mode for our pattern matching semantics, we introduce two more execution constructs to simplify formalizing the additional constraints they impose. The first is the *trail table* (T), which simplifies enforcing the `DIFFERENT EDGES` matching mode constraint across the entire pattern list. Similar to binding tables, trail tables are also bags of records, but each record is also annotated with a set of graph edges ($\mathcal{W}$) bound in that record. So edge-disjointness across trail tables corresponding to different path patterns in a pattern list can be trivially enforced by checking the intersection of the bound-edge sets of their respective records. The second is the *path table* (P), for simplifying enforcing the `TRAIL` constraint when matching path patterns. Unlike pattern lists, the bindings in a single path pattern match correspond to a single path in the working graph. So the path table builds on top of the trail table entries by annotating them with the graph nodes corresponding to the endpoints of the matched path. This simplifies path composition to a trivial endpoint continuity check, while the same bound-edge sets from the trail tables can be reused for the `TRAIL` constraint.

Definition 5.2 (Trail Tables). A *trail table entry* is a tuple $(\mathsf{R}, \mathcal{W})$ of a record R and the edges $\mathcal{W} \in \mathcal{P}(\mathcal{E})$ bound in that record, i.e., a bound-edge set. A *trail table* T is a bag of trail table entries. We equip trail tables with a commutative monoid $(\mathsf{T},\ \bowtie,\ \mathbf{1}_\mathsf{T})$, where $\mathbf{1}_\mathsf{T} \triangleq \{\!\{([\,], \varnothing)\}\!\}$; for implicitly enforcing edge-disjointness when joining them. The *trail product* $\otimes$ combines two trail table entries only when their records join and their bound-edge sets are disjoint, and $\bowtie$ lifts it to trail tables:

$$\mathsf{T}_1 \bowtie \mathsf{T}_2 \triangleq \left\{\!\!\left\{ t_1 \otimes t_2 \,\middle|\, \begin{array}{l} t_1 \in \mathsf{T}_1,\ t_2 \in \mathsf{T}_2, \\ t_1 \otimes t_2 \neq \bot \end{array} \right\}\!\!\right\} \qquad (\mathsf{R}_1, \mathcal{W}_1) \otimes (\mathsf{R}_2, \mathcal{W}_2) \triangleq \begin{cases} (\mathsf{R}_1 \bowtie \mathsf{R}_2, \mathcal{W}_1 \cup \mathcal{W}_2) & \text{if } \mathsf{R}_1 \bowtie \mathsf{R}_2 \neq \bot \textbf{ and } \\ & \mathcal{W}_1 \cap \mathcal{W}_2 = \varnothing \\ \bot & \text{otherwise} \end{cases}$$

Definition 5.3 (Path Tables). A *path entry* is a tuple $(\mathsf{R}, \mathsf{n}_{\mathtt{lft}}, \mathsf{n}_{\mathtt{rht}}, \mathcal{W})$ containing a record, the matched path's endpoint nodes, and its bound-edges. A *path table* P is a bag of path entries. Path tables compose under $\diamond_\oplus$, which implicitly enforces endpoint continuity and edge-disjointness. We parameterize it with the operation $\oplus$ that defines how to combine the records of the two entries:

$$\mathsf{P}_1 \diamond_\oplus \mathsf{P}_2 \triangleq \left\{\!\!\left\{ (\mathsf{R}_1 \oplus \mathsf{R}_2,\ \mathsf{n}_{\mathtt{lft1}},\ \mathsf{n}_{\mathtt{rht2}},\ \mathcal{W}_1 \cup \mathcal{W}_2) \,\middle|\, \begin{array}{l} (\mathsf{R}_i,\ \mathsf{n}_{\mathtt{lft}i},\ \mathsf{n}_{\mathtt{rht}i},\ \mathcal{W}_i) \in \mathsf{P}_i\ (i \in \{1,2\}), \\ \mathsf{R}_1 \oplus \mathsf{R}_2 \neq \bot, \quad \mathsf{n}_{\mathtt{rht1}} = \mathsf{n}_{\mathtt{lft2}}, \quad \mathcal{W}_1 \cap \mathcal{W}_2 = \varnothing \end{array} \right\}\!\!\right\}.$$

The parametrization is important because although the semantics of path composition remains unchanged between quantified and unquantified path patterns (it only depends on endpoint continuity and edge-disjointness), how variables may bind to nodes/edges differs. Unquantified variables cannot bind to more than one node/edge per record, while quantified variables may bind to multiple depending on the quantifier used. Parametrization allows us to enforce these differences by defining how records combine during path composition. *Path concatenation* $\diamond \triangleq \diamond_\bowtie$ instantiates $\oplus$ with the natural join of records, while Definition 5.4 defines it for quantified path patterns.

Definition 5.4 (Quantified Path Repetition). Let $\mathcal{Y} = \mathbf{vars}(P)$ be the variables declared by a quantified path and let K be its quantifier. The quantifier repetition is bounded below by $\mathbf{lo}(K)$ (Figure 4) and above by $\mathbf{hi}_{\mathcal{G}}(K)$ (Figure 5), which caps the unbounded quantifiers by the number of graph edges (`TRAIL` constraint of edge-disjointness). Variables declared inside the quantifier accumulate bindings by the *repetition extension* $\diamond_K \triangleq \diamond_{\bigstar_K}$, starting from the *zero-repetition table* $\mathbf{0}^K_{\mathcal{Y}}$; for $x \in \mathcal{Y}$:

$$\mathbf{0}^K_{\mathcal{Y}} \triangleq \left\{\!\!\left\{ \left(\left[x \mapsto \begin{cases} \textsc{Null} & \text{if } K = ? \\ [] & \text{else} \end{cases} \right]_{x \in \mathcal{Y}}, \mathrm{n}, \mathrm{n}, \varnothing \right) \,\middle|\, \mathrm{n} \in \mathcal{N} \right\}\!\!\right\} \qquad (\mathrm{R}_1 \bigstar_K \mathrm{R}_2)(x) \triangleq \begin{cases} \mathrm{R}_2(x) & \text{if } K = ? \\ \mathrm{R}_1(x) :: [\mathrm{R}_2(x)] & \text{else} \end{cases}$$

The resulting group representation is the runtime counterpart of the quantifier lift $(\cdot)\uparrow_K$ (Definition 4.2). It is list-valued for every quantifier other than ?, which admits a single repetition and keeps the binding itself on successful match, or **Null** otherwise. The zero-repetition table is a path table pairing every node in the graph with itself, so joining it with P under $\diamond_K$ leaves endpoints and bound-edge sets unchanged and only lifts each record into its group representation. Therefore, $\mathbf{0}^K_{\mathcal{Y}} \diamond_K \mathrm{P}$ is the group representation of one repetition, and repeating it $\kappa - 1$ times further, yields the matches that use exactly κ repetitions.

5.2 Pattern Semantics

Pattern matching is responsible for introducing bindings into the query pipeline. A **Match** clause enumerates every homomorphism of its pattern into the working graph, which the standard specifies as a multi-phase pipeline (ISO §22.3)—each path pattern is evaluated independently, followed by forcing agreement on shared variables via natural equijoins on the cross products of their binding tables, producing the final result table. We formalize pattern matching using three main judgements.

$$\mathcal{G} \vdash_{\mathsf{Atom}} A \Downarrow \mathrm{B} \qquad \mathcal{G} \vdash_{\mathsf{Pat}} \hat{P} \to \hat{P}' \qquad \mathcal{G} \vdash_{\mathsf{PatLst}} \hat{p} \to \hat{p}'$$

The *atom judgement* matches a pattern atom A, i.e., a node N or an edge E atom, in isolation to produce a binding table B of singleton records. It is big-step ($\Downarrow$) as it only depends on the working graph. The *path judgement* reduces path patterns (Figure 5) to path tables with endpoint continuity and edge-disjointness enforced. The *pattern list judgement* reduces pattern lists to trail tables by composing path patterns using conjunction (p, P), while enforcing edge-disjointness throughout.

Atom Matching (Level 1). The first step in atom matching is to determine if the labels of a graph element ι (node or edge) satisfy the label expression l. We use an auxiliary judgement $\iota \vdash_\Lambda l \Downarrow b$ for this, which evaluates to a Boolean $b \in \{\textsc{True}, \textsc{False}\}$ indicating whether the labels of ι satisfy l.

$$\textsc{L-Empty}\ \frac{}{\iota \vdash_\Lambda \varepsilon \Downarrow \textsc{True}} \qquad \textsc{L-Atom}\ \frac{L \in \Lambda(\iota)}{\iota \vdash_\Lambda L \Downarrow \textsc{True}} \qquad \textsc{L-Atom-Fail}\ \frac{L \notin \Lambda(\iota)}{\iota \vdash_\Lambda L \Downarrow \textsc{False}}$$

$$\textsc{L-Wild}\ \frac{\Lambda(\iota) \neq \varnothing}{\iota \vdash_\Lambda \,\% \Downarrow \textsc{True}} \qquad \textsc{L-Wild-Fail}\ \frac{\Lambda(\iota) = \varnothing}{\iota \vdash_\Lambda \,\% \Downarrow \textsc{False}}$$

$$\textsc{L-Neg}\ \frac{\iota \vdash_\Lambda l \Downarrow b}{\iota \vdash_\Lambda \,!l \Downarrow \neg b} \qquad \textsc{L-And}\ \frac{\iota \vdash_\Lambda l_1 \Downarrow b_1 \qquad \iota \vdash_\Lambda l_2 \Downarrow b_2}{\iota \vdash_\Lambda l_1 \,\&\, l_2 \Downarrow b_1 \wedge b_2} \qquad \textsc{L-Or}\ \frac{\iota \vdash_\Lambda l_1 \Downarrow b_1 \qquad \iota \vdash_\Lambda l_2 \Downarrow b_2}{\iota \vdash_\Lambda l_1 \,|\, l_2 \Downarrow b_1 \vee b_2}$$

The rules follow the same structure as their counterparts in atom typing (§4.2), which filter schemas.

$$\textsc{Atom-Node}\ \frac{\mathrm{B} := \left\{\!\!\left\{ [\mathbf{var}(N) \mapsto \mathrm{n}] \,\middle|\, \begin{array}{l} \mathrm{n} \in \mathcal{N}, \\ \mathbf{prp}(N) \subseteq \Xi(\mathrm{n}), \\ \mathrm{n} \vdash_\Lambda \mathbf{lbl}(N) \Downarrow \textsc{True} \end{array} \right\}\!\!\right\}}{\mathcal{G} \vdash_{\mathsf{Atom}} N \Downarrow \mathrm{B}} \qquad \textsc{Atom-Edge}\ \frac{\mathrm{B} := \left\{\!\!\left\{ [\mathbf{var}(E) \mapsto \mathrm{e}] \,\middle|\, \begin{array}{l} \mathrm{e} \in \mathcal{E}, \mathbf{prp}(E) \subseteq \Xi(\mathrm{e}), \\ \mathrm{e} \vdash_\Lambda \mathbf{lbl}(E) \Downarrow \textsc{True}, \\ \Lambda(\mathrm{e}) \in \mathbf{dir}(\mathbf{ornt}(E)) \end{array} \right\}\!\!\right\}}{\mathcal{G} \vdash_{\mathsf{Atom}} E \Downarrow \mathrm{B}}$$

Atom-Node collects every graph node whose property map is a superset of the atom's property map and whose labels satisfy its label expression, binding each of them to $\mathbf{var}(N)$ in a singleton record. Atom-Edge is the symmetric rule for edges, with an additional predicate for directionality.

Path Pattern Matching (Level 2). Path pattern matching (ISO §16.7) reduces a path pattern to a path table (Definition 5.3), leveraging the latter's implicit enforcement of endpoint continuity and edge-disjointness constraints for simplifying the formalization of the former's semantic rules.

$$\textsc{Path-Node}\ \frac{\mathcal{G} \vdash_{\text{Atom}} N \Downarrow \mathrm{B}}{\mathcal{G} \vdash_{\text{Pat}} N \to \boldsymbol{\eta}_{\mathcal{G},N}(\mathrm{B})} \qquad \textsc{Path-Edge-L}\ \frac{\mathcal{G} \vdash_{\text{Pat}} \hat{P} \to \hat{P}'}{\mathcal{G} \vdash_{\text{Pat}} \hat{P}\,E\,N \to \hat{P}'\,E\,N} \qquad \textsc{Path-Edge-R}\ \frac{\mathcal{G} \vdash_{\text{Pat}} N \to \mathrm{P}_2}{\mathcal{G} \vdash_{\text{Pat}} \mathrm{P}_1\,E\,N \to \mathrm{P}_1\,E\,\mathrm{P}_2}$$

$$\textsc{Path-Edge}\ \frac{\mathcal{G} \vdash_{\text{Atom}} E \Downarrow \mathrm{B}}{\mathcal{G} \vdash_{\text{Pat}} \mathrm{P}_1\,E\,\mathrm{P}_2 \to \mathrm{P}_1 \diamond \boldsymbol{\eta}_{\mathcal{G},E}(\mathrm{B}) \diamond \mathrm{P}_2}$$

Path-Node lifts the binding table (B) with singleton records from a node atom (N) match into a path table with empty bound-edges, i.e., $\mathcal{W} = \varnothing$; using the metafunction $\boldsymbol{\eta}_{\mathcal{G},N}(\mathrm{B})$ defined in Figure 5. Path-Edge-L and Path-Edge-R reduce the two sides of $\hat{P}\,E\,N$. Path-Edge applies once both are path tables by lifting the binding table of the edge atom (E) using the edge-symmetrical metafunction $\boldsymbol{\eta}_{\mathcal{G},E}(\mathrm{B})$, and then joining the path tables via path concatenation $\diamond$. This is the runtime counterpart of the endpoint condition $\boldsymbol{\theta}_E(\cdot,\cdot,\cdot)$ from §4.2; handling edge orientation via $\mathbf{ends}_{\mathcal{G},E}(\mathrm{e})$.

$$\textsc{Path-Concat-L}\ \frac{\mathcal{G} \vdash_{\text{Pat}} \hat{P}_1 \to \hat{P}_1{}'}{\mathcal{G} \vdash_{\text{Pat}} \hat{P}_1 \circ \hat{P}_2 \to \hat{P}_1{}' \circ \hat{P}_2} \qquad \textsc{Path-Concat-R}\ \frac{\mathcal{G} \vdash_{\text{Pat}} \hat{P}_2 \to \hat{P}_2{}'}{\mathcal{G} \vdash_{\text{Pat}} \mathrm{P}_1 \circ \hat{P}_2 \to \mathrm{P}_1 \circ \hat{P}_2{}'} \qquad \textsc{Path-Concat}\ \frac{\mathrm{P} := \mathrm{P}_1 \diamond \mathrm{P}_2}{\mathcal{G} \vdash_{\text{Pat}} \mathrm{P}_1 \circ \mathrm{P}_2 \to \mathrm{P}}$$

$$\textsc{Path-QPath-Step}\ \frac{\mathcal{G} \vdash_{\text{Pat}} \hat{P} \to \hat{P}'}{\mathcal{G} \vdash_{\text{Pat}} [\hat{P}]^K_{\mathcal{Y}} \to [\hat{P}']^K_{\mathcal{Y}}} \qquad \textsc{Path-QPath}\ \frac{\mathcal{G} \vdash_{\text{QPat}} \hat{K} \to \hat{K}'}{\mathcal{G} \vdash_{\text{Pat}} \hat{K} \to \hat{K}'}$$

Path-Concat-L, Path-Concat-R, and Path-Concat reduce path patterns connected by GQL's concatenation operator $\circ$. This operator allows connecting path patterns at their endpoint node atoms by enforcing that the variables corresponding to these endpoint node atoms must agree, i.e., are the same. We do not consider this in the source calculus (Figure 2) but introduce it in the runtime calculus (Figure 5) for conveniently desugaring quantified edges into quantified path patterns with anonymous (variable declaration absent) endpoint node atoms. Path-QPath-Step similarly steps the path inside a quantifier frame $[\cdot]^K_{\mathcal{Y}}$ while preserving the frame itself, and Path-QPath delegates it to the auxiliary quantified path judgement once the path pattern is fully reduced to a path table.

GQL's quantified paths are parallel to regular-expression repetition. Their semantics from the standard can be interpreted using our execution constructs, as the union $\mathrm{P}^{\mathbf{lo}(K)} \cup \cdots \cup \mathrm{P}^{\mathbf{hi}_{\mathcal{G}}(K)}$ of the path tables obtained by iterating the inner path table P between the quantifier's bounds. We compute this union with a frontier. A quantified path runtime term $[\mathrm{P}]^K_{\mathcal{Y}}\langle \mathrm{P}_A, \mathcal{F}, \kappa\rangle$ carries the *accumulator* P_A of the matched bindings, accumulated starting from repetition $\mathbf{lo}(K)$ to repetition $\kappa - 1$; while the *frontier* $\mathcal{F}$ contains only those bindings matched using exactly κ repetitions.

$$\textsc{QPath-Init}\ \frac{\mathrm{P}_A := \text{if } \mathbf{lo}(K) = 0 \text{ then } \mathbf{0}^K_{\mathcal{Y}} \text{ else } \{\!\{\}\!\} \qquad \mathcal{F} := \mathbf{0}^K_{\mathcal{Y}} \diamond_K \mathrm{P}}{\mathcal{G} \vdash_{\text{QPat}} [\mathrm{P}]^K_{\mathcal{Y}} \to [\mathrm{P}]^K_{\mathcal{Y}}\langle \mathrm{P}_A, \mathcal{F}, 1\rangle} \qquad \textsc{QPath-Finish}\ \frac{\kappa > \mathbf{hi}_{\mathcal{G}}(K) \ \vee\ \mathcal{F} = \{\!\{\}\!\}}{\mathcal{G} \vdash_{\text{QPat}} [\mathrm{P}]^K_{\mathcal{Y}}\langle \mathrm{P}_A, \mathcal{F}, \kappa\rangle \to \mathrm{P}_A}$$

$$\textsc{QPath-Iter}\ \frac{\kappa \le \mathbf{hi}_{\mathcal{G}}(K) \qquad \mathcal{F} \neq \{\!\{\}\!\} \qquad \mathrm{P}_{A'} := \text{if } \mathbf{lo}(K) \le \kappa \text{ then } \mathrm{P}_A \uplus \mathcal{F} \text{ else } \mathrm{P}_A \qquad \mathcal{F}' := \text{if } \kappa < \mathbf{hi}_{\mathcal{G}}(K) \text{ then } \mathcal{F} \diamond_K \mathrm{P} \text{ else } \{\!\{\}\!\}}{\mathcal{G} \vdash_{\text{QPat}} [\mathrm{P}]^K_{\mathcal{Y}}\langle \mathrm{P}_A, \mathcal{F}, \kappa\rangle \to [\mathrm{P}]^K_{\mathcal{Y}}\langle \mathrm{P}_{A'}, \mathcal{F}', \kappa + 1\rangle}$$

QPath-Init initiates the iteration by computing the one-repetition frontier from the zero-repetition table by extending it with the inner path table (Definition 5.4), followed by instantiating the accumulator path table P_A with the zero-repetition table itself iff the quantifier lower bound $\mathbf{lo}(K)$

is zero. This enables supporting GQL's *empty match* feature for quantifiers. QPath-Iter applies as long as the counter tracking the number of repetitions κ is not greater than the quantifier upper bound $\mathbf{hi}_{\mathcal{G}}(K)$, and as long as the current frontier $\mathcal{F}$ is not empty. In each iteration, it accumulates the current frontier $\mathcal{F}$ into the accumulator P_A (if $\kappa \geq \mathbf{lo}(K)$), and then computes the next frontier $\mathcal{F}'$ by extending the current one with the inner path table P. Since all extensions happen using the repetition extension operator $\diamond_K$, endpoint continuity and edge-disjointness are implicitly enforced.

Pattern List Matching (Level 3). Pattern list matching composes path patterns via conjunction (p, P), whose cross product followed by equijoin semantics is captured by our trail-monoid's join $\bowtie$ (Definition 5.2). It joins on the shared variables while enforcing `DIFFERENT EDGES` at the same time.

$$\textsc{PatLst-Lift-Step}\ \frac{\mathcal{G} \vdash_{\mathsf{Pat}} \hat{P} \to \hat{P}'}{\mathcal{G} \vdash_{\mathsf{PatLst}} \hat{P} \to \hat{P}'} \qquad \textsc{PatLst-Lift}\ \frac{\mathrm{T} := \boldsymbol{\pi}_{\mathrm{T}}(\mathrm{P})}{\mathcal{G} \vdash_{\mathsf{PatLst}} \mathrm{P} \to \mathrm{T}}$$

$$\textsc{PatLst-Conj-L}\ \frac{\mathcal{G} \vdash_{\mathsf{PatLst}} \hat{p} \to \hat{p}'}{\mathcal{G} \vdash_{\mathsf{PatLst}} \hat{p}, \hat{P} \to \hat{p}', \hat{P}} \qquad \textsc{PatLst-Conj-R}\ \frac{\mathcal{G} \vdash_{\mathsf{PatLst}} \hat{P} \to \hat{P}'}{\mathcal{G} \vdash_{\mathsf{PatLst}} \mathrm{T}_1, \hat{P} \to \mathrm{T}_1, \hat{P}'}$$

$$\textsc{PatLst-Conj}\ \frac{\mathrm{T} := \mathrm{T}_1 \bowtie \mathrm{T}_2}{\mathcal{G} \vdash_{\mathsf{PatLst}} \mathrm{T}_1, \mathrm{T}_2 \to \mathrm{T}}$$

PatLst-Lift-Step lifts a single path pattern step into the pattern list judgement, and PatLst-Lift lowers the reduced path table of a path pattern into a trail table by projecting only the record and bound-edge set attributes via $\boldsymbol{\pi}_{\mathrm{T}}(\cdot)$ from Figure 5. PatLst-Conj-L and PatLst-Conj-R reduce either side of a conjunction left-to-right, while PatLst-Conj joins their trail tables when fully reduced.

5.3 Query Semantics

Formalizing query evaluation semantics is yet again relatively simple since most of the complexity has been delegated to value expression and pattern matching semantics, with only the formalization of the flow of the binding tables between query clauses remaining. We use two judgement forms:

$$\underbrace{\Omega \vdash \hat{Q} \to \hat{Q}'}_{\text{Linear Query}} \qquad \underbrace{\Omega \vdash_{\circledast} \hat{q} \to \hat{q}'}_{\text{Composite Query}}$$

The *linear query judgement* rewrites linear queries in the context of the database catalog Ω into one of two intermediate tuples: (1) $\langle \mathcal{G}, \hat{p}, \phi, \mu \rangle$ —for formalizing **Match** and **Where** clause evaluation, and (2) $\langle \mathcal{G}, \mathrm{B}, \mu \rangle$—for formalizing **Return** clause evaluation. The *composite query judgement* formalizes composite query evaluation where the semantics depend on the composite operator $\circledast$.

Linear Query Evaluation. Focused linear queries first select a working graph, then match a pattern against it, optionally filter the matched bindings, before projecting to generate the result.

$$\textsc{Q-Use}\ \frac{\mathrm{G} \in \mathbf{dom}(\Omega) \qquad \mathcal{G} = \Omega(\mathrm{G}) \qquad \hat{p} := \mathbf{norm}(p)}{\Omega \vdash \textbf{Use}\ \mathrm{G}\ \textbf{Match}\ p\ \textbf{Where}\ \phi\ \textbf{Return}\ \mu \to \langle \mathcal{G}, \hat{p}, \phi, \mu \rangle} \qquad \textsc{Q-Match}\ \frac{\mathcal{G} \vdash_{\mathsf{PatLst}} \hat{p} \longrightarrow^* \mathrm{T}}{\Omega \vdash \langle \mathcal{G}, \hat{p}, \phi, \mu \rangle \to \langle \mathcal{G}, \mathrm{T}, \phi, \mu \rangle}$$

$$\textsc{Q-Where}\ \frac{\mathrm{B} := \left\{\!\!\left\{ \mathrm{R} \in \boldsymbol{\pi}_{\neg \mathcal{A}_{\mathsf{Anon}}}(\boldsymbol{\pi}_{\mathrm{B}}(\mathrm{T})) \mid \mathcal{G};\ \mathrm{R} \vdash \phi \longrightarrow^* \textsc{True} \right\}\!\!\right\}}{\Omega \vdash \langle \mathcal{G}, \mathrm{T}, \phi, \mu \rangle \to \langle \mathcal{G}, \mathrm{B}, \mu \rangle}$$

$$\textsc{Q-Return}\ \frac{\mathrm{B}' := \left\{\!\!\left\{ [x_i \mapsto \omega_i] \;\middle|\; \begin{array}{l} \mathrm{R} \in \mathrm{B},\ \forall (\vartheta_i\ \textbf{As}\ x_i) \in \mu. \\ \mathcal{G};\ \mathrm{R} \vdash \vartheta_i \longrightarrow^* \omega_i \end{array} \right\}\!\!\right\}}{\Omega \vdash \langle \mathcal{G}, \mathrm{B}, \mu \rangle \to \mathrm{B}'}$$

Q-Use resolves the graph G against the catalog Ω and normalizes the pattern list of the **Match** clause using $\mathbf{norm}(\cdot)$ (Figure 5), which expands its quantified edges and names its anonymous pattern atoms, before rewriting the query to an intermediate tuple. Only the rule for queries with a **Where** clause is shown since those without can be rewritten into one. Q-Match reduces the pattern list to a trail table using the multi-step closure $\longrightarrow^*$ of the pattern list judgement. Q-Where filters this trail table by lowering it to a binding table via $\boldsymbol{\pi}_{\mathrm{B}}(\cdot)$ from Figure 5, before dropping the variables introduced by $\mathbf{norm}(\cdot)$ using $\boldsymbol{\pi}_{\neg \mathcal{A}_{\mathsf{Anon}}}(\cdot)$ (as they cannot be referenced in any clause);

and finally retaining only the records for which the predicate reduces to **True**. Q-Return projects the optionally filtered binding table using the projection list μ to generate the final binding table.

Composite Query Evaluation. Composite queries compose linear queries via composite operators.

$$\textsc{CQ-Lift}\ \frac{\Omega \vdash \hat{Q} \to \hat{Q}'}{\Omega \vdash_{\circledast} \hat{Q} \to \hat{Q}'} \qquad \textsc{CQ-L}\ \frac{\Omega \vdash_{\circledast} \hat{q} \to \hat{q}'}{\Omega \vdash_{\circledast} \hat{q} \circledast \hat{Q} \to \hat{q}' \circledast \hat{Q}} \qquad \textsc{CQ-R}\ \frac{\Omega \vdash_{\circledast} \hat{Q} \to \hat{Q}'}{\Omega \vdash_{\circledast} \mathrm{B}_1 \circledast \hat{Q} \to \mathrm{B}_1 \circledast \hat{Q}'} \qquad \textsc{CQ-Union}\ \frac{}{\Omega \vdash_{\circledast} \mathrm{B}_1\ \textsc{Union}\ \mathrm{B}_2 \to \mathrm{B}_1 \uplus \mathrm{B}_2}$$

CQ-Lift lifts linear query stepping into composite query judgement. CQ-L and CQ-R reduce the operands left-to-right until both operands are binding tables. CQ-Union computes bag union preserving multiplicities. Since the judgement is indexed by the composite operator $\circledast$, the congruence rules thread the same operator through both operands, enforcing the GQL standard's requirement that composite queries use a single composite operator across all their linear queries.

6 Type Soundness

We prove *type soundness*, i.e., every well-formed query produces a binding table when evaluated to completion; that conforms to the schema assigned to the query statically by the typing rules. The typing judgments assign types to *source terms*, while the semantics reduce *runtime terms* containing intermediate forms. Auxiliary conformance and configuration-typing relations bridge this gap.

Definition 6.1 (Value Typing and Conformance). A runtime value $\omega \in \mathcal{V}$ *inhabits* a type τ, written $\omega : \tau$, when the value is a member of the type's semantic domain: scalars inhabit their base type, **Null** inhabits ? and every nullable type $\tau?$, graph elements inhabit their graph-indexed identity types (up to subtyping), and lists inhabit **List** τ when every element does. Value typing is closed under subtyping: $\omega : \tau$ and $\tau <: \tau_2$ imply $\omega : \tau_2$. A record R then conforms to a record schema Γ, written $\mathrm{R} \models \Gamma$, when $\mathbf{dom}(\mathrm{R}) = \mathbf{dom}(\Gamma)$ and $\forall x \in \mathbf{dom}(\Gamma).\mathrm{R}(x) : \tau.\ \tau <: \Gamma(x)$, and a binding or trail table conforms to Γ when every record in its support does. A path table P conforms to Γ when every entry $(\mathrm{R}, \mathrm{n}_{\mathtt{lft}}, \mathrm{n}_{\mathtt{rht}}, \mathcal{W})$ has $\mathrm{R} \models \Gamma$, $\mathrm{n}_{\mathtt{lft}}, \mathrm{n}_{\mathtt{rht}} \in \mathcal{N}$, and $\mathcal{W} \subseteq \mathcal{E}$; we write $\cdot \models \Gamma$ for all of these.

Configuration Typing. A database world $\Sigma = (\Omega, \Upsilon)$ is *well-formed*, written $\models \Sigma$, when every closed graph site's instance conforms to its declared graph schema (Definition 2.3). *Configuration typing*—written $\Sigma; \mathrm{G} \vdash_{\mathrm{pcfg}} \hat{p} : \Gamma_{\mathrm{rt}} \Rightarrow \Gamma$ for a pattern configuration and $\Sigma \vdash_{\mathrm{cfg}} \hat{q} : \Gamma$ for a query configuration—closes the source typing rules under the runtime forms of Figure 5, requiring every table a configuration has materialized to conform to the internal schema of the subterm it replaced, and the clauses that remain to compose that schema into Γ.

Theorem 6.1 (Expression Soundness). Suppose $\models \Sigma$, $\mathcal{G} = \Omega(\mathrm{G})$, and $\mathrm{R} \models \Gamma$.

(1) *(Progress)* If $\vartheta \notin \mathcal{V}$, then there exists ϑ' such that $\mathcal{G}; \mathrm{R} \vdash \vartheta \to \vartheta'$.
(2) *(Soundness)* If $\Sigma; \mathrm{G}; \Gamma \vdash^{\square}_{\diamond} \vartheta : \tau \triangleright \mathcal{X}$ is derivable for some $\square$, $\diamond$, $\mathcal{X}$, and $\mathcal{G}; \mathrm{R} \vdash \vartheta \longrightarrow^{*} \omega$ with $\omega \in \mathcal{V}$, then $\omega : \tau_1$ for some $\tau_1 <: \tau$.

Proof sketch. Progress: every computation rule has a complementary null rule, so a non-value either has a reducible operand or contracts. Soundness: by induction on ϑ, splitting the reduction sequence into subterm evaluations along the left-to-right congruence rules. Computation rules return values at the operator's result type, while the null rules—which fire exactly when an operand's dynamic refinement fails at runtime—return **Null** at type ?. Both are subtypes of the nullable types the rules assign, which is why the conclusion is up to subtyping. □

Theorem 6.2 (Pattern Configuration Safety). Suppose $\models \Sigma$, $\mathcal{G} = \Omega(\mathrm{G})$, and $\Sigma; \mathrm{G} \vdash_{\mathrm{pcfg}} \hat{p} : \Gamma_{\mathrm{rt}} \Rightarrow \Gamma$.

(1) *(Progress)* Either $\hat{p} = \mathrm{T}$, or there exists $\hat{p}'$ such that $\mathcal{G} \vdash_{\mathsf{PatLst}} \hat{p} \rightarrow \hat{p}'$.
(2) *(Preservation)* If $\mathcal{G} \vdash_{\mathsf{PatLst}} \hat{p} \rightarrow \hat{p}'$, then $\Sigma; \mathrm{G} \vdash_{\mathsf{pcfg}} \hat{p}' : \Gamma_{\mathrm{rt}} \Rightarrow \Gamma$.
(3) *(Terminal conformance)* If $\hat{p} = \mathrm{T}$, then $\boldsymbol{\pi}_{\neg \mathcal{A}_{\mathsf{Anon}}}(\boldsymbol{\pi}_{\mathrm{B}}(\mathrm{T})) \models \Gamma$.

Proof sketch. By induction on the configuration-typing derivation. The path cases use atom conformance and closure of the path-table operations under compatible record joins, and the conjunction case trail-join conformance and its schema-compatibility premise; trail and endpoint checks only remove entries. The quantified case uses an inner induction on κ: the zero-repetition table $\mathbf{0}^K_{\mathcal{Y}}$ inhabits the lifted schema, and each application of $\diamond_K$ preserves it since $\bigstar_K$ extends a list-typed (or, for ?, nullable) binding by one element of the inner schema's type. Terminal conformance then follows since $\boldsymbol{\pi}_{\mathrm{B}}(\cdot)$ and $\boldsymbol{\pi}_{\neg \mathcal{A}_{\mathsf{Anon}}}(\cdot)$ only erase annotations/attributes that Γ_{rt} adds. □

Corollary 6.1 (Pattern Soundness). If $\models \Sigma$, $\mathcal{G} = \Omega(\mathrm{G})$, $\Sigma; \mathrm{G} \vdash_{\mathsf{PatLst}} p : \Gamma$, and $\mathcal{G} \vdash_{\mathsf{PatLst}} \mathbf{norm}(p) \longrightarrow^* \mathrm{T}$, then $\boldsymbol{\pi}_{\neg \mathcal{A}_{\mathsf{Anon}}}(\boldsymbol{\pi}_{\mathrm{B}}(\mathrm{T})) \models \Gamma$.

Proof sketch. Normalization typing supplies the internal schema Γ_{rt}; iterated preservation and terminal conformance from Theorem 6.2 then give the result. □

Theorem 6.3 (Query Type Soundness). If $\models \Sigma$, $\Sigma \vdash Q : \Gamma$, and $\Omega \vdash Q \longrightarrow^* \mathrm{B}$, then $\mathrm{B} \models \Gamma$.

Proof sketch. Progress and preservation for configurations, then iteration. *Progress*: well-formedness of Σ lets a source query resolve its graph site; a pattern-stage configuration steps its pattern list, and transitions to a 3-tuple once that list has reduced to a trail table and every predicate to a value—both terminate, since each pattern rule either reduces a subterm to a table or advances a counter capped by $\mathbf{hi}_{\mathcal{G}}(K) \leq |\mathcal{E}|$, and each expression step strictly decreases the number of non-value subterms; a 3-tuple projects (by Theorem 6.1); a final bag is terminal. *Preservation*: graph resolution (Q-Use) produces a well-typed pattern-stage configuration; pattern stepping (Q-Match) preserves it by Theorem 6.2; filtering (Q-Where) lowers the trail table, erases the generated attributes—unreferenceable, since no source query mentions an attribute in $\mathcal{A}_{\mathsf{Anon}}$—and selects a sub-bag, so terminal conformance gives a 3-tuple well-typed at Γ_1; and projection (Q-Return) applies Theorem 6.1 per expression. Iterating yields $\Sigma \vdash_{\mathsf{cfg}} \mathrm{B} : \Gamma$, i.e., $\mathrm{B} \models \Gamma$. □

Corollary 6.2 (Composite Query Soundness). If $\models \Sigma$, $\Sigma \vdash_{\circledast} q : \Gamma$, and $\Omega \vdash_{\circledast} q \longrightarrow^* \mathrm{B}$, then $\mathrm{B} \models \Gamma$.

Proof sketch. By induction on the composite-query typing derivation. The base case is Theorem 6.3; in the inductive case CQ-Lift, CQ-L, and CQ-R reduce the operands to bags conforming to their operand schemas, and every composite operator either merges the two bags or selects one of them, so the result conforms to the composite schema that CompQry-Compose assigns. □

7 Mechanized Implementation

We implemented a complete mechanization in Lean 4 comprising more than 23,000 lines across 14 modules, following the GQL calculus in Figure 2. Table 1 summarizes the module layout; the core formalization and its soundness proofs (excluding tests and benchmarks) total roughly 21,300 lines.

The sort hierarchy is encoded as inductive types: `BaseSort` for T_0, `ExtSort` for T_1 (including graph-scoped and schema-refined types), and `GSort` for full sorts τ, while values use a mutual inductive (`Value`/`ValueList`) for kernel-derived `DecidableEq` over nested lists. Subtyping is an inductive proposition with 17 constructors matching the paper's rules; S-Union-Congruence is derived as a theorem from primitives due to a Lean kernel restriction on nested inductives. The typing rules of Section 4 are encoded as eleven inductive propositions totaling 57 constructors across expression, predicate, property-constraint, atom, refinement, pattern, pattern-expression, projection, projection-list, query, and single-operator composite typing.

The semantics module provides executable definitions for all reduction rules, including direction-aware endpoint conditions, frontier-based quantified path iteration, and set operations with duplicate elimination. Beyond the executable definitions, the mechanization proves the full soundness development of Section 6 (`Metatheory.lean`), the small-step layer with progress, preservation, and a proven-equivalent deterministic interpreter behind a user-facing engine flag (`SmallStep.lean`), and a certified executable type checker (`TypeChecker.lean`): `inferQuery` computes a result schema for any query in the fragment, `inferQuery_sound` shows every accepted query is well typed in the declarative system, and composition with query type soundness guarantees that an accepted query's result table conforms to the inferred schema. All results depend only on the standard Lean axioms (`propext`, `Classical.choice`, `Quot.sound`), with zero `sorry` and zero user-declared axioms.

Table 1. Module structure of the Lean 4 mechanization.

Module	Formalization	LoC
`Sorts.lean`	Sec. 3 ($\mathcal{T}_0$, $\mathcal{T}_1$, $\mathcal{T}_2$)	254
`Values.lean`	Sec. 2.1 (values, records)	362
`Syntax.lean`	Fig. 2 (expr., patterns, queries)	291
`Schema.lean`	Def. 2.1–2.3 (property graphs)	290
`RecordSchema.lean`	Def. 3.2–3.4, Sec. 4 (record ops)	375
`Subtyping.lean`	Sec. 3.5 (subtyping)	297
`Typing.lean`	Sec. 4 (typing judgments)	1,133
`Semantics.lean`	Sec. 5 (executable semantics)	638
`SmallStep.lean`	Sec. 5 (step relations)	2,267
`Metatheory.lean`	Sec. 6 (soundness proofs)	14,057
`TypeChecker.lean`	Sec. 4 (executable checker)	1,364
`Test.lean`	Unit tests (276 assertions)	1,252
`Examples.lean`	Worked examples (12 files)	404
`LDBCBench.lean`	LDBC SNB integration tests	671
Total		**23,655**

The artifact's tests are layered: 276 unit assertions cover each computational unit and rule, 30 worked examples mirror the paper's definitions one by one, including negative cases, and the six expressible queries of the LDBC SNB Interactive v2 workload [22], translated into GQL as `Query` terms, form an end-to-end integration test: each is type checked by the certified checker, evaluated against golden results, checked to conform to its inferred schema, and executed on both engines with bit-for-bit agreement. Table 2 summarizes the queries from the SNB workload, evaluated and type checked using MGQL. In total, the build checks 350 `native_decide` assertions. Supporting other queries requires features outside the considered fragment with most leaning towards GQL's relational core, i.e., data transformations.

Table 2. Sample LDBC SNB queries type checked and evaluated using MGQL.

Query	Pattern	Features
IS1	1 directed edge	label, prop constraint
IS3	1 undirected edge	undirected direction
IS4	single node	prop constraint
IS5	1 directed edge	label filter
IC8	3-hop chain	conjunction, left dir.
IC2	2-hop + WHERE	conj., WHERE filter

8 Related Work

Graph Query Languages. Early graph query formalisms centered on RDF and SPARQL [18, 21, 23, 24], which operate on triple-based data and require indirect encodings for the richer metadata found in property graphs. Traversal-based languages such as Gremlin [25] offer fine-grained navigational control at the expense of declarative reasoning. In contrast, pattern-matching languages such as Cypher [15], PGQL [30], and G-CORE [2]; provide a more declarative interface. GQL [11, 20] is the first standardized graph query language for property graphs, standardized as ISO/IEC 39075 in 2024. It unifies ideas from its predecessors and incorporates graph schemas as part of the language specification. Yet the standard's entirely informal specification of GQL's semantics, is not conducive to formal reasoning; leading to potential inconsistencies and ambiguities when implementing it, or even in the standard's specification itself. Francis et al. [13] distilled GQL's pattern matching into a Graph Pattern Calculus (GPC) equipped with typing rules and a denotational semantics under set semantics, and Gheerbrant et al. [16] defined Core GQL and Core PGQ as concise formal models focused on understanding its expressivity. Our work is complementary to these efforts. We

provide a small-step operational semantics, a schema-aware type system with a machine-checked soundness proof; all under bag semantics. None of these are covered by GPC or Core GQL.

Language Mechanization. Mechanizing language specifications in proof assistants has offered several cross-domain benefits. Bodin et al. [7] mechanized ECMAScript 5 in Coq, producing both a formal semantics and an extracted reference interpreter. De Santo et al. [10] gave a comprehensive Coq mechanization of JavaScript regular expressions per ECMA-262, uncovering errors in previous formalisms. Seassau et al. [28] formalized a substantial OCaml subset in Rocq. But GQL's specification has not been mechanized before, making MGQL the first to mechanize it and thus providing the first formal foundation for rigorously reasoning about its semantics.

Formal Semantics of Query Languages. Relational query languages have been formalized extensively. Guagliardo and Libkin [17] gave a formal semantics for a large SQL fragment, while Benzaken and Contejean [6] and Chu et al. [8] provided mechanized Coq treatments of SQL with nulls, aggregations, and bag semantics. For graph query languages, Regular Path Queries and their extensions [5] primarily focus on providing a clean compact theoretical foundation, leaving out several key features of GQL, Cypher, etc., such as properties, schemas, and bag semantics. Formal semantics exist for core Cypher constructs [3, 15], and Ye et al. [31] proposed a gradual-typing calculus for GQL's path patterns, but none of these are machine-checked. Díaz et al. [12] mechanized GraphQL in Coq, but GraphQL is a tree-shaped API language and does not contain a graph pattern matching fragment. Our work is the first mechanized formalization of GQL in a proof assistant (Lean4).

9 Conclusion

We have presented MGQL, a formal semantics for a substantial fragment of the ISO/IEC 39075 Graph Query Language. Our development provides three interlocking contributions: a type system that exploits graph schemas for static refinement while tracking three-valued nullability and heterogeneous union types; a small-step operational semantics that makes explicit the interplay among Kleene three-valued logic, trail-aware pattern composition, quantified-path iteration, and the clause-by-clause query pipeline; and a layered type soundness argument establishing that well-typed queries produce results conforming to their declared schemas. MGQL provides the first bridge between GQL's specification and a mechanized implementation.

Acknowledgments

We thank Cheng Ding, Ivan Grigorik, Linghan Zhong, Lara Marinov, and the anonymous reviewers for helpful feedback and discussions. This work was supported in part by the U.S. National Science Foundation (NSF) Nos. CCF-2217696, CCF-2313027, CCF-2403036; and an Amazon Research Award (Fall 2025). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Amazon.

Data Availability Statement

We mechanized our small-step formalization of the semantics of GQL in Lean4. The artifact supporting this paper is available on Zenodo [29]. It also contains a big-step semantics formalization.

References

[1] Renzo Angles. 2018. The property graph database model. In *AMW*.

[2] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD*. 1421–1432. doi:10.1145/3183713.3190654

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. In *CSUR*. 1–40. doi:10.1145/3104031

[4] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. Pg-schema: Schemas for property graphs. *PACMMOD* (2023), 1–25. doi:10.1145/3589778

[5] Pablo Barceló Baeza. 2013. Querying graph databases. In *PODS*. 175–188. doi:10.1145/2463664.2465216

[6] Véronique Benzaken and Évelyne Contejean. 2019. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *CPP*. 249–261. doi:10.1145/3293880.3294107

[7] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *POPL*. 87–100. doi:10.1145/2578855.2535876

[8] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*. 510–524. doi:10.1145/3140587.3062348

[9] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *CADE*. 625–635. doi:10.1007/978-3-030-79876-5_37

[10] Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. 2024. A Coq Mechanization of JavaScript Regular Expression Semantics. *ICFP* (2024), 1003–1031. doi:10.1145/3674666

[11] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD*. 2246–2258. doi:10.1145/3514221.3526057

[12] Tomás Díaz, Federico Olmedo, and Éric Tanter. 2020. A mechanized formalization of GraphQL. In *CPP*. 201–214. doi:10.1145/3372885.3373822

[13] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. 2023. GPC: A Pattern Calculus for Property Graphs. In *PODS*. 241–250. doi:10.1145/3584372.3588662

[14] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. 2023. A Researcher's Digest of GQL. In *ICDT*. doi:10.4230/LIPIcs.ICDT.2023.1

[15] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *SIGMOD*. 1433–1445. doi:10.1145/3183713.3190657

[16] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. GQL and SQL/PGQ: Theoretical Models and Expressive Power. *Proc. VLDB Endow.* 18, 6 (2025), 1798–1810. doi:10.14778/3725688.3725707

[17] Paolo Guagliardo and Leonid Libkin. 2017. A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.* 11, 1 (2017), 27–39. doi:10.14778/3151113.3151116

[18] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 Query Language. W3C Recommendation.

[19] ISO/IEC. 2023. ISO/IEC 9075-2:2023: Information technology — Database languages SQL — Part 2: Foundation (SQL/Foundation). International Standard. https://www.iso.org/standard/76584.html

[20] ISO/IEC. 2024. *ISO/IEC 39075:2024 — Information technology — Database languages — GQL*. Technical Report. International Organization for Standardization.

[21] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with Property Paths. In *ISWC*. 3–18. doi:10.1007/978-3-319-25007-6_1

[22] Linked Data Benchmark Council (LDBC). 2023. LDBC SNB Interactive v2 Implementations. https://github.com/ldbc/ldbc_snb_interactive_v2_impls.

[23] Katja Losemann and Wim Martens. 2013. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.* 38, 4 (2013). doi:10.1145/2494529

[24] Jorge Perez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. In *TODS*. 1–45. doi:10.1145/1567274.1567278

[25] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *DBPL*. 1–10. doi:10.1145/2815072.2815073

[26] Marko A. Rodriguez and Peter Neubauer. 2010. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology* 36, 6 (2010), 35–41. doi:10.1002/bult.2010.1720360610

[27] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71. doi:10.1145/3434642

[28] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025. Formal Semantics and Program Logics for a Fragment of OCaml. In *ICFP*. 128–159. doi:10.1145/3747509

[29] Aditya Thimmaiah, Tong-Nong Lin, and Milos Gligoric. 2026. *MGQL: An Executable, Small-Step Semantics of GQL (artifact)*. doi:10.5281/zenodo.21669636

[30] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES*. 1–6. doi:10.1145/2960414.2960421

[31] Wenjia Ye, Matías Toro, Tomás Díaz, Bruno C. d. S. Oliveira, Manuel Rigger, Claudio Gutierrez, and Domagoj Vrgoč. 2025. Flexible and Expressive Typed Path Patterns for GQL. In *OOPSLA*. 302–328. doi:10.1145/3763061

Received 2026-03-18; accepted 2026-06-10