Flexible and Expressive Typed Path Patterns for GQL*

WENJIA YE, National University of Singapore, Singapore
MATÍAS TORO, University of Chile & IMFD, Chile
TOMÁS DÍAZ, University of Chile, Chile
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China
MANUEL RIGGER, National University of Singapore, Singapore
CLAUDIO GUTIERREZ, Universidad de Chile & IMFD, Chile
DOMAGOJ VRGOČ, Pontificia Universidad Católica de Chile & IMFD, Chile

Graph databases have become an important data management technology across various domains, including biology, sociology, industry (*e.g.* fraud detection, supply chain management, financial services), and investigative journalism, due to their ability to efficiently store and query large-scale knowledge graphs and networks. Recently, the Graph Query Language (GQL) was introduced as a new ISO standard providing a unified framework for querying graphs. However, this initial specification lacks a formal type system for query validation. As a result, queries can fail at runtime due to type inconsistencies or produce empty results without prior warning. Solving this issue would help users write correct queries, especially on large datasets.

To address this gap, we introduce a formal type model for a core fragment of GQL extended with property-based filtering and imprecise types both in the schema and the queries. This model, named FPPC, enables static detection of semantically incorrect and stuck queries, improving user feedback. We establish key theoretical properties, including *emptiness* (detecting empty queries due to type mismatches) and *type safety* (guaranteeing that well-typed queries do not fail at runtime). Additionally, we prove a *gradual guarantee*, ensuring that removing type annotations either does not introduce static type errors or only increases the result set. By integrating imprecision into GQL, FPPC offers a flexible solution for handling schema evolution and incomplete type information. This work contributes to making GQL more robust, improving both its usability and its formal foundation.

CCS Concepts: • **Theory of computation** → **Type structures**; • **Information systems** → **Query languages**; • **Software and its engineering** → **Semantics**.

Additional Key Words and Phrases: GQL, gradual typing, property graphs

ACM Reference Format:
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. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 283 (October 2025), 27 pages. https://doi.org/10.1145/3763061

*This research was supported by a Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant, ANID DFG 220011, ANID Fondecyt Regular Grant Nr. 1240346, and the Millennium Science Initiative Program: code ICN17_002.

Authors' Contact Information: Wenjia Ye, National University of Singapore, Singapore, Singapore, wenjiaye@nus.edu.sg; Matías Toro, PLEIAD Lab, Computer Science Department (DCC), University of Chile & IMFD, Santiago, Chile, mtoro@dcc.uchile.cl; Tomás Díaz, Computer Science Department (DCC), University of Chile, Santiago, Chile, tdiaz@dcc.uchile.cl; Bruno C. d. S. Oliveira, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk; Manuel Rigger, National University of Singapore, Singapore, Singapore, rigger@nus.edu.sg; Claudio Gutierrez, Computer Science Department (DCC), Universidad de Chile & IMFD, Santiago, Chile, cgutierr@dcc.uchile.cl; Domagoj Vrgoč, Pontificia Universidad Católica de Chile & IMFD, Santiago, Chile, vrdomagoj@uc.cl.

ACM 2475-1421/2025/10-ART283
https://doi.org/10.1145/3763061

1 Introduction

Graph databases have been a steadily growing technology since the early 2000s and are commonly used in areas such as Biology [Redaschi and Consortium 2009], the pharmaceutical industry,[1] or investigative journalism [Balalau et al. 2024]. Most recently graph databases gained major popularity due to their adoption as the platform to store and manage large Knowledge Graphs [Hogan et al. 2021] and through their incorporation into Retrieval Augmented Generation (RAG) methods to model the underlying expert knowledge. Graphs offer an intuitive modelling of the domain of choice, with nodes representing entities and edges various relations these entities can form.

Early adoption of the technology was mostly driven by the Semantic Web community, resulting in the RDF data format [Cyganiak et al. 2014] and the SPARQL query language [Harris et al. 2013], which are both W3C standards. RDF modelling takes a simple approach to data, basically viewing it as edge labelled graphs [Angles and Gutiérrez 2008]. On the other hand, commercial systems [Memgraph Team 2023; TigerGraph Team 2021; Vesoft Inc/Nebula 2023; Webber 2012] throughout the years used the more involved property graph data model, where both nodes and edges can have labels and a series of attribute-value pairs attached to them. The majority of commercial systems also deploy their own approach to querying such data making interoperability and data sharing a major issue. To rectify the situation, the ISO standardization body started working on a common language for querying property graphs, resulting in the recently published Graph Query Language (GQL) standard [ISO/IEC 2024]. GQL provides a standardized way to query these graphs, building on previous graph query languages like Cypher [Francis et al. 2023a].

GQL queries can be viewed as sets of path patterns, which are expressions with variables that specify a sequence of nodes and edges to be matched by a path in a property graph. Figure 1 illustrates a simple property graph representing a social network, adapted from the benchmark proposed by Angles et al. [2023]. In this graph, nodes represent people, who may be teachers or students, as well as comments posted in a course forum. For example, the node n_1 has two labels, `Person` and `Teacher`, and is described via its two attributes: `name` and `status`. Similarly, edges model relationships such as likes, authorship of comments, and knows relationships between two people. For instance, the edge e_3 tells us that Bob is the author of the comment modelled by the node n_3. Consider a query that retrieves all the teachers who liked a comment made by students who they know since 2021 or earlier. This can be expressed by the following GQL query pattern:

```
(x:Teacher)-[:Likes]->(:Comment)-[:Author]->(:Student)-[y:Knows WHERE
                                                   y.since < 2021]-(x)
```

Here we are binding a `:Teacher`-labelled node to the variable `x` and requiring there be a `:Likes`-labelled edge to a node representing a `Comment`. In GQL ASCII-art syntax, round brackets represent nodes, while square brackets represent edges. We then require this comment to have been made by a student (represented by an `:Author` edge to a node with the label `Student`). Finally, we require this student to have any edge to the original node bound to `x`, that this edge is labelled with `:Knows` and that the value of its `since` attribute is greater than 2021.

As with most query languages, a key challenge in GQL is efficient query evaluation. While most graph queries (particularly those involving long path patterns) require a large number of joins to traverse relationships, adding a type system on top of GQL would allow to provide early feedback on potential failures, reducing unnecessary computations, and improving query reliability. Additionally, this would also help in steering users away from writing semantically incorrect queries. To illustrate this benefit, consider again the social network query described previously. Suppose the final node in the path is specified via the node pattern `(x WHERE x.status > 1)` instead

[1] https://www.ontotext.com/knowledgehub/case-bstudies/astrazeneca-bboosted-bearly-bhypotheses-btesting-bby-busing-bontotext-blld-binentory/

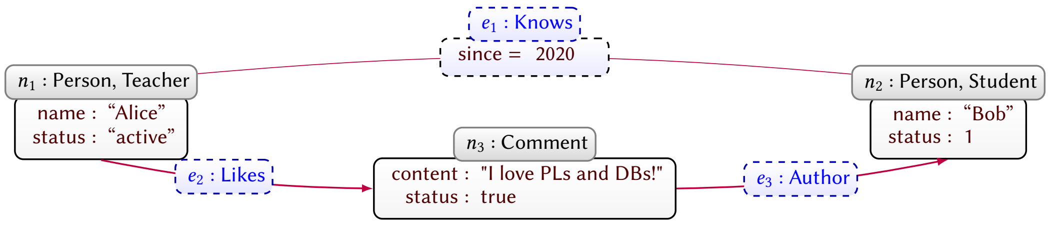

Fig. 1. A sample property graph database.

of simply as `(x)`. Because this node must match the first node, which is a `Teacher`, and the `status` property for teachers is a `String`, the comparison with an integer is guaranteed to fail. This means the query would always produce an empty result. A type system can detect this issue before execution, allowing the user to correct the query and avoid unnecessary computation.

Typing for GQL has been explored in prior work. The Graph Pattern Calculus (GPC) [Francis et al. 2023a], provides a formal model for the path pattern language of GQL but does not use schema information for validation. Additionally, GPC only provides a coarse-grained type system that does not track labels or property types for nodes and edges, limiting its ability to prevent execution errors or detect queries that return empty results. Similarly, schema definition approaches, which do introduce some form of typing, such as the recently proposed PG-Schema [Angles et al. 2023], explain the conceptual approach to rooting out ill-formed queries. However, they do not go in-depth of how to incorporate this information into GQL or its fragments. Despite these efforts, a significant gap remains. The GQL ISO standard does not specify typing rules for queries. While the standard defines types for data, it lacks a formal type system for validating queries. As a result, queries can fail at runtime due to type inconsistencies or return empty results without prior warning.[2]

In this paper, we address this gap by designing a type system driven by the following three goals:

1. **Identify patterns that always yield empty results**. The type system should issue warnings for queries that are guaranteed to return no matches due to type inconsistencies.
2. **Prevent patterns that may cause execution failure**. The type system should reject queries that are deemed syntactically invalid according to the ISO standard, or which could result in stuck evaluations or runtime errors. Examples include `(x)-[x]->`, which attempts to bind the same variable as both a node and an edge, and `(x WHERE y.status > 1)`, where `y` is not bound.
3. **Support flexible type definitions**. The type system should enable a gradual adoption of types, allowing programmers to incrementally refine the type annotations and improve precision. This flexibility is particularly beneficial for databases with evolving schemas or incomplete type information. For instance, when using no schema (a fully-imprecise schema), the query pattern `(x: Teacher WHERE x.status > 0)` is accepted statically but returns an empty list if no `Teacher` node has an integer `status` property. By refining the schema to specify that `Teacher` nodes have a `status` property of type `String`, then the query becomes ill-typed.

To achieve these goals, we introduce the Flexible Path Pattern Calculus (FPPC), a formal model of the path pattern language of GQL extended with new syntax for filtering by property types.[3] To achieve flexibility, we draw inspiration from *gradual typing* [Siek and Taha 2006]. We extend both the schema and path pattern language with imprecise type information by introducing an

[2]This was also identified as an issue by the standardization committee.
[3]This is needed because, like all usual standards documents, the GQL ISO standard is written in a mix of prose and pseudocode, making it difficult to interpret and apply in a rigorous, research-driven manner. Its descriptions are lengthy (500+ pages) and prone to misinterpretation, resembling a legal document rather than a precise formal specification.

unknown type ⋆ (which represents any label or property), allowing queries to be type-checked even when the type schema information is imprecise.

In summary, this paper makes the following contributions:

- We present FPPC, a formal calculus for GQL that extends the ISO standard with property-based filtering and gradual types. Our approach enables users to incrementally adopt types in both the schema and pattern queries while maintaining close alignment with the standard.
- We extend the path pattern language with *type property expressions*, allowing nodes and edges to be filtered based on both labels and property types.
- We prove the *emptiness* property (§6.2), ensuring that if a query type checks but its type is flagged as empty (according to an emptiness predicate), it will definitively return an empty result due to type inconsistencies.
- We prove *type safety* of the type system (§6.1), guaranteeing that well-typed pattern queries cannot get stuck or produce runtime errors.
- We show that types are entirely optional. Moreover, we prove a *gradual guarantee* property (§6.3), which ensures that removing type annotations does not introduce either type or runtime errors. This result is established by analyzing the interaction between casts and type tests and with adjustments to the precision relation to maintain this property.
- We provide a prototype implementation of FPPC in Python (§7), which supports parsing, type checking, and executing path patterns over concrete graph databases. The type system rejects queries that may get stuck and provides warnings for potential inconsistencies that lead to empty results.

While our approach is primarily motivated by the Graph Query Language (GQL), the underlying principles are general enough to be applied to similar scenarios, such as regular path queries (RPQs), or property graphs in general. All proofs and a prototype are in the supplementary materials.

2 Overview

This section gives an overview of FPPC, a formal calculus for the path pattern language of GQL. We introduce GQL, outlining its core constructs and query evaluation model. Next, we discuss issues that can arise in path patterns, such as inconsistencies and type mismatches. We then present FPPC in action, explaining how it extends the GQL path pattern language with gradual typing and property-based filtering. Next, we give a high-level description of key properties that ensure the correctness and reliability of the system. Finally, we outline its current limitations.

2.1 GQL in a Nutshell

The graph datamodel of GQL consists of one or more *property graphs*, comprised of a set of *nodes* and *edges*, both directed and undirected, which represent relationships between nodes. Nodes and edges are defined by a unique identifier, a set of labels and some properties. Fig. 1 shows an example graph database representing a simple social network, where the solid boxes denote nodes, while the arrows and dashed boxes represent edges. The first node on the left, for instance, has identifier n_1, a single label `Account`, and two properties indicating that the `name` is `"Alice"` and that the `status` is `"active"`. Edges follow a similar structure but may also be directed. Here, edges e_2 and e_3 point from node n_1 to n_3 and from n_3 to n_2, respectively, whereas edge e_1 is undirected.

GQL can be used to query graph databases, analogous to how SQL is used for relational databases. A basic GQL query is shown below:

```
USE SocialNetwork
MATCH (x WHERE x.status=true)-[z:Author]->(y)
RETURN y.name AS student_name, x.content AS comment_content
```

The first line selects the database to query, namely `SocialNetwork`, and the second one specifies a path pattern to match in the graph. The last line defines what to return from the matched pattern.

This work focuses on the second line, specifically in path patterns that yield a table containing intermediate results up to that point of evaluation, such as when there are multiple pattern paths in a query. Path patterns are the main building block of GQL queries, and they closely resemble regular path queries (RPQs) [Barceló et al. 2014].

The path pattern in line 2 consists of three subpatterns, listed from left to right as they appear in the query. The first one matches any node, noted by `( )`, where the `status` property is `true`, storing identifiers in variable `x`. The second subpattern matches directional edges, captured by `-[ ]->`, that have an `Author` label, and binding them to variable `z`. Finally, the third one matches any node, storing identifiers in `y`. Each of these subpatterns yields a set of *paths*, where a path is defined as an alternating sequence of nodes and edges, starting and ending with a node. In particular, edge patterns yield the edge's identifier and include the connected nodes that form the path.

Once all patterns are evaluated, their results are *concatenated* (if possible) yielding a new set of concatenated paths. Two paths are concatenable if the last node of the first matches the first node of the second. In our example, the subpatterns yield the singleton sets of paths $\{n_3\}$, $\{n_3 e_3 n_2\}$ and the sets of paths $\{n_1, n_2, n_3\}$, respectively. Their concatenation produces the following intermediate table (on the left), where `x`, `y` and `z` correspond to the variables bound in the pattern. To complete the GQL query, the third line projects the properties `name` and `content` from nodes `y` (n_2) and `x` (n_3), yielding the final table (on the right):

path	x	y	z
$n_3 e_3 n_2$	n_3	n_2	e_3

student_name	comment_content
“Bob”	“I love PLs and DBs!”

2.2 Errors in Path Patterns

As in any other language, GQL queries can contain errors. These errors can be classified into syntactic errors and semantic errors [Brass and Goldberg 2006]. A syntactic error occurs when the query does not conform to the language's grammar. A semantic error occurs when the query is syntactically valid but does not produce the intended result.

Given that GQL was only recently standardized [ISO/IEC 2024], there is limited empirical research on common mistakes in GQL queries. However, studies on semantic errors in SQL suggest that similar issues are likely to arise in GQL [Jiang et al. 2024]. Furthermore, the nature of graph queries introduces additional complexity, particularly in pattern matching and variable scoping. To illustrate, we categorize several types of errors that can occur in GQL queries and discuss their causes and consequences.

Incompatible Shapes. Consider the pattern `(x)-[x]->`, which attempts to match any node followed by any edge. This pattern is invalid because `x` is used to represent both a node and an edge, which are distinct entities. The GQL ISO standard treats such patterns as syntactically incorrect.

Incompatible Types. The following pattern:

```
(x WHERE x.status is bool)-[:Knows]-(x WHERE x.status is str)
```

attempts to match edges connecting a node to itself, as indicated by the use of the binding `x` in both nodes. The issue that arises in this pattern is that the property `status` is expected to be a boolean in one case and a string in the other. Although this pattern is valid according to the ISO standard, it always results in an empty output, as a value cannot simultaneously be a boolean and a string.

Nonexistent Attributes. Consider the pattern `(x WHERE x.stauts > 0)`, which contains a typo: `stauts` instead of `status`. If no node in the database has the property `stauts`, the pattern's behavior is ambiguous. The ISO standard suggests propagating an error in such cases, but it also allows an alternative interpretation where the condition evaluates to false or null, yielding an empty result.

Incorrect Attribute Usage. If the database defines `name` as a string, then the pattern `(x WHERE x.name > 0)` incorrectly applies a numeric comparison to a string. The ISO standard suggests raising an error, but implementations may instead treat the condition as false, yielding an empty result.

Unbound Variables. Patterns like `(y WHERE x.status = true)` refer to an unbound variable `x`. In this case, the query should terminate with a fatal error, as `x` has not been assigned a value.

These examples highlight the importance of a robust type system in GQL. As discussed earlier (§1), a major challenge in GQL query evaluation is the large number of joins required to traverse relationships. Early detection of inconsistent or malformed queries, such as those listed above, reduces unnecessary computation and improves efficiency.

2.3 Types in GQL

GPC [Francis et al. 2023a] was developed when the ISO standard for GQL was still in draft form. Since then, the published standard has introduced several changes. The main objective of GPC was to distill the core components of the draft into a formal calculus. It introduces several simplifications, such as restricting patterns to single-label expressions and assuming that constants belong to the same family of values. Its primary focus is on path queries, and it includes a basic type system aimed at avoiding syntactic errors. The type system defines atomic types such as Node, Edge, and Path, as well as inductive types like Maybe, for optional values, and Group, for lists of types. Given a path query, the type rules assign a type to a single variable. This type system allows GPC to reject errors such as incompatible types and unbound variables, as described in §2.2.

The ISO standard, officially published in early 2024 [ISO/IEC 2024], after GPC, introduces the concept of a *graph schema*, which is particularly relevant to this work. A *graph type descriptor* is defined as a set of *node types* N and a set of *edge types* E. Unlike GPC, where node types are minimal, the ISO standard defines them more richly as a set of one or more labels and a set of zero or more property types. A property type consists of a name identifier and a value type. Edge types are similarly defined, comprising a set of zero or more labels, a set of possibly empty property types, two (possibly identical) endpoint node types, an indication of the edge's directionality, and source and target node types if the edge is directional. Value types in the ISO standard resemble those in SQL, including booleans, numbers, strings, and dates. Additionally, they support open dynamic union types, such as $T_1 \mid T_2$. The top type, ANY, represents all types.

While the ISO standard provides a structured way to define graph schemas, it leaves open the question of how to systematically type-check queries. The following section introduces a type system for GQL that builds on ideas from GPC and incorporates the richer type definitions from the ISO standard, extending these foundations with additional features.

2.4 FPPC **in Action**

The type system of FPPC assigns types to path patterns, yielding types for each bound variable and the overall path. Unlike previous approaches, this system takes a schema as input to exclude paths that cannot match entities in the database. However, since many graph databases lack predefined schemas, FPPC facilitates adoption by allowing schemas to include unknown types, denoted as $\star$. These unknown types can represent any possible type depending on the context, allowing users to gradually refine a fully imprecise schema into a more precise one, or conversely to generalize specific type information when flexibility is needed.

For example, the type $(\!|\texttt{Person}\ \{\texttt{name}: \texttt{String}, \texttt{status}: \texttt{String}\}|\!)$ characterizes a node (N) with the label (ℓ) `Person` and exactly two properties (R): a string `name` and a string `status`. When applied to the graph of Fig. 1, this type applies to n_1, but not to n_2 or n_3. To include n_2, the following type can be used: $(\!|\texttt{Person}\ \{\texttt{name}: \texttt{String}, \star\}|\!)$, where $\{\texttt{name}: \texttt{String}, \star\}$ represents an *open record* R and allows for an unbound number of additional properties. If no type information is available, the fully imprecise type $(\!|\star\ \{\star\}|\!)$ (abbreviated as $(\!|\star|\!)$) can be used to represent a node with any label and any set of properties.

Following the ISO standard, edge types are composed of four components: two node types, a directionality, and edge information (labels and properties). For example, the edge type $(\!|\star|\!) \xrightarrow{\texttt{Author}} (\!|\star|\!)$ represents an (imprecise) edge with the label `Author` that connects any two nodes. Similarly, we use $(\!|\star|\!) \overset{\texttt{Knows}\ \{\texttt{since}:\ \mathbb{Z}\}}{\sim\sim} (\!|\star|\!)$ to represent an undirected edge with the label `Knows` and property `since` of type integer. One can start with a fully imprecise schema $S_I = (\!|\star|\!), (\!|\star|\!) \xrightarrow{\star} (\!|\star|\!), (\!|\star|\!) \overset{\star}{\sim} (\!|\star|\!)$, representing any node, and any directional or undirected edge. From there, one may transition to a more precise schema, such as $S_P = N_P, N_C, N_C \xrightarrow{\texttt{Author}} N_P, N_P \overset{\texttt{Knows}\ \{\texttt{since}:\ \mathbb{Z}\}}{\sim\sim} N_P, N_P \xrightarrow{\texttt{Likes}} (\!|\star|\!)$, where $N_P = (\!|\texttt{Person}\ \{\texttt{name}: \texttt{String}, \star\}|\!)$, and $N_C = (\!|\texttt{Comment}\ \{\star\}|\!)$. Eventually, the schema can be made fully precise: $S_F = N_T, N_S, N_C, N_C \xrightarrow{\texttt{Author}} N_S, N_T \overset{\texttt{Knows}\ \{\texttt{since}:\ \mathbb{Z}\}}{\sim\sim} N_S, N_T \xrightarrow{\texttt{Likes}} N_C$, where $N_T = (\!|\texttt{Person\&Teacher}\ \{\texttt{name}: \texttt{String}, \texttt{status}: \texttt{String}\}|\!)$, $N_S = (\!|\texttt{Person\&Student}\ \{\texttt{name}: \texttt{String}, \texttt{status}: \mathbb{Z}\}|\!)$, $N_C = (\!|\texttt{Comment}\ \{\texttt{content}: \texttt{String}, \texttt{status}: \mathbb{B}\}|\!)$, and $A\&B$ denotes a conjunction, meaning that the node has both labels A and B.

Adopting types, gradually. Given a schema and a path pattern, the type system either yields an error (over-approximating stuck evaluations), or produces a *path type* (P) and a *type environment* (Γ), which can be used to under-approximate empty results. A path type P corresponds to the type of the path itself, matching the *path* column in the working table from §2.1. Intuitively, a path type is a sequence of node types N that may be returned by the pattern and is used to predict concatenation errors, which is why only node information is relevant. A type environment Γ assigns a type to each variable in the path pattern and corresponds to the remaining columns of that table.

Extending the path pattern language. Additionally, FPPC extends element pattern filtering with *is property expressions*, allowing direct filtering of nodes and edges based on both labels and property types. For example, the pattern `(y : ★ {name: str, ★})` filters all nodes that have any label and a `name` property of type string. While this notation does not add expressive power—it can be rewritten using the ISO-defined syntax `(y WHERE PROPERTY_EXISTS(y, name) AND y.name IS STRING)`—it is more concise and improves type consistency detection during type checking.[4]

Consider the schema S_P that describes the graph of Fig. 1 and the following path pattern:

```
(x : {status : bool} WHERE x.status = true)-[z : Author]->(y)
```

The type system returns path type $N_C - N_A$ where $N_C = (\!|\texttt{Comment}\ \{\texttt{status}: \mathbb{B}, \star\}|\!)$ and $N_A = (\!|\texttt{Person}\ \{\texttt{name}: \texttt{String}, \star\}|\!)$, indicating that the path starts with a node labelled `Comment` and property `status`, and ends at a `Person` node that has a `name` attribute. It also returns the type environment $(x \mapsto N_C + N_P, y \mapsto N_C + N_A, z \mapsto N_C \xrightarrow{\texttt{Author}} N_A)$, where $A + B$ denotes a gradual union type, meaning either A or B.

The type system of FPPC can detect all errors from §2.2. Let us revisit the examples to illustrate how the type system works.

[4]Such queries can be transformed into the proposed notation through type inference on conditional expressions. Alternatively, additional support could be provided by maintaining a mapping between labels and property types. These considerations, however, are orthogonal to this work and are left for future research.

Example 2.1. Pattern `(x)-[x]->` is ill-typed. Indeed, sub-pattern `(x)` has type environment $\Gamma_1 = x \mapsto (\!|\star|\!)$ whereas sub-pattern `-[x]->` has type environment $\Gamma_2 = x \mapsto (\!|\star|\!) \xrightarrow{\star} (\!|\star|\!)$. When type checking the concatenation pattern, both type environments are concatenated as well by computing the *meet* between both environments. Intuitively, this is done by computing the most precise information for each variable. In this case the meet between a node type and an edge type is not defined, and then the pattern is rejected.

Example 2.2. Pattern `(x : {status: bool})->(x : {status: String})` is well-typed but triggers an empty-result warning. This is because the node sub-patterns yield type environments $\Gamma_1 = x \mapsto (\!|\star\ \{\texttt{status} : \mathbb{B}\}|\!)$ and $\Gamma_2 = x \mapsto (\!|\star\ \{\texttt{status} : \texttt{String}\}|\!)$, respectively, which yield the environment $x \mapsto (\!|\star\ \{\texttt{status} : \bot\}|\!)$ when concatenated, because the meet between $\mathbb{B}$ and `String` is $\bot$. In this case, a bottom type means that during runtime there will be no node that has a property that is both a boolean and a string at the same time. Notice that the presence of bottom does not guarantee an empty result. For example, the pattern `(x : {status:` $\mathbb{B}$`})->(x : {status: String+`$\mathbb{B}$`})` yields environment $x \mapsto (\!|\star\ \{\texttt{status} : \bot + \mathbb{B}\}|\!)$, meaning that x is either empty or a boolean, thus possibly non-empty.

Example 2.3. Pattern `(x: {stauts: int} WHERE x.stauts > 0)` is well-typed, but may raise an empty-result warning. With the fully imprecise schema S_I, variable x has type $(\!|\star\ \{\texttt{stauts} : \star\}|\!)$, whereas with schema $(\!|\star\ \{\texttt{status} : \mathbb{B}\}|\!), \ldots$, x is typed as $\bot$, raising an empty-result warning.

Example 2.4. Similarly, pattern `(x: {status: bool} WHERE x.status > 0)` is well-typed, but raises an empty-result. The reason is that the path type now is $\bot$, since the type error in the conditional term is propagated to the pattern. Similar to the second example, union `(x: {status: bool} WHERE x.status > 0) + ()` does not raise an empty-result warning, as it can match either a non-empty result or an empty one.

Example 2.5. Finally, the pattern `(y WHERE x.status = true)` is ill-typed. This is determined by checking whether the variable x is present in the type environment. Since only the variable y is in scope, the pattern is correctly rejected.

3 Preliminaries: Data Model

Before introducing FPPC, following GPC [Francis et al. 2023a], this section presents the formal model of graphs G. This model uses several countable and disjoint sets: node identifiers $\mathcal{N}$, directed edge identifiers $\mathcal{E}_d$, undirected edge identifiers $\mathcal{E}_u$, label identifiers $\mathbb{L}$, key identifiers $\mathcal{K}$, and a set of constants Const. A property graph is formally defined as the tuple $G = \langle \mathcal{N}_{id}, \mathcal{E}_d, \mathcal{E}_u, \mathcal{L}, \text{endpoints}, \text{src}, \text{tgt}, \mathcal{P} \rangle$, where $\mathcal{L} : N_{id} \cup E_d \cup E_u \to 2^{\mathbb{L}}$ is a labeling function that assigns each identifier a (possibly empty) finite set of labels from $\mathbb{L}$, $\text{src}, \text{tgt} : E_d \to N$ define the source and target nodes of a directed edge, $\text{endpoints} : E_u \to 2^{\mathcal{N}}$ defines the endpoints of undirected edges, $\delta : (N_{id} \cup E_d \cup E_u) \to \mathcal{K} \rightharpoonup \text{Const}$ is a partial function that associates a constant with an identifier and a key from $\mathcal{K}$.

For instance, for the graph of Figure 1, the graph G can be defined using records as partial functions as follows: $\mathcal{N}_{id} = \{n_1, n_2, n_3\}, \mathcal{E}_d = \{e_2, e_3\}, \mathcal{E}_u = \{e_1\}, \mathcal{L} = \{n_1 : \{\texttt{Person}, \texttt{Teacher}\}, n_2 : \{\texttt{Person}, \texttt{Student}\}, n_3 : \{\texttt{Comment}\}\}, \text{endpoints} = \{e_1 : \{n_1, n_2\}\}, \text{src} = \{e_2 : n_1, e_3 : n_3\}, \text{tgt} = \{e_2 : n_3, e_3 : n_2\}, \delta = \{n_1 : \{\texttt{name} : \text{“Alice”}, \{\texttt{status} : \text{“active”}\}, \ldots\}$.

Finally, we introduce the notion of paths, which represent sequences of nodes and edges in the result of path patterns. A path p is an ordered sequence of alternating nodes and edges $n_1 e_1 n_2 e_2 n_3 \ldots$ such that for all i, either (1) $\text{src}(e_i) = n_i$ and $\text{tgt}(e_i) = n_{i+1}$, meaning e_i is a forward edge, (2) $\text{src}(e_i) = n_{i+1}$ and $\text{tgt}(e_i) = n_i$, meaning e_i a backward edge, or (3) $\text{endpoints}(e_i) = \{n_i, n_{i+1}\}$, meaning e_i is an undirected edge).

Base Types	$\iota ::= \mathbb{Z} \mid \mathbb{B} \mid \mathsf{String}$
Simple Types	$\tau ::= \iota \mid \star \mid \tau + \tau \mid \bot$
Attributes	$a \in \{\mathtt{name}, \mathtt{status}, a, b, \ldots\}$
Property Types	$R ::= \{\overline{a_i : \tau_i}^{i}\} \mid \{\overline{a_i : \tau_i}^{i}, \star\} \mid \bot$
Labels	$\mathtt{l} \in \{\mathtt{Person}, \mathtt{Student}, A, B, \ldots\}$
Label Types	$\ell ::= \mathtt{l} \mid \star \mid \ell \& \ell \mid \ell + \ell \mid \varepsilon$
Descriptor Types	$L ::= \ell\, R$
Constants	$c ::= z \mid s \mid b$
Expressions	$t ::= c \mid t \;\mathsf{bop}\; t \mid \mathsf{uop}\; t \mid x.a \mid t \;\mathsf{as}\; \tau \mid t \;\mathsf{is}\; \tau$
Optional Variables	$x? ::= x \mid \_$
Descriptors	$d ::= x? : L$
Directions	$\Longleftrightarrow ::= \leftarrow \mid \rightarrow \mid \sim \mid -$
Path Patterns	$\pi ::= (d) \mid \overset{d}{\Longleftrightarrow} \mid \pi + \pi \mid \pi\ \pi \mid \pi_{\langle t \rangle} \mid \pi^{l..u}$

Fig. 2. Syntax of Patterns.

4 Dynamic Semantics

In this section, we introduce the dynamic semantics of FPPC, extending the semantics of GPC in three main aspects: runtime filtering now involves both labels and property types (previously, only labels were considered); gradual typing is included to allow flexible path patterns, and conditioned expressions are enhanced to support more complex query structures. We start by defining the syntax and then formally define the extended semantics below.

4.1 Syntax

Figure 2 present the syntax of patterns. Compared to GPC, our approach adds explicit type matching for properties. To ensure a smooth transition from the untyped nature of standard GQL to a fully typed system, we introduce the gradual type $\star$ as a placeholder for missing or unknown type information, applicable to both labels, and property types. This design enables flexibility on label matching and incremental adoption of property matching, while ensuring compatibility with existing GQL queries.

The core component of path patterns is the concept of *descriptors* d, which define atomic matching conditions for nodes and edges. Each descriptor consists of an optional variable for binding matched entities and a descriptor type. In GPC a descriptor type is a single label, whereas in FPPC, a descriptor type is generalized to *label types* ℓ and *property types* R. Label types include concrete labels ($\mathtt{l}$), such as `Person` or `Student`; unknown labels ($\star$), matching any label; intersection labels ($\ell\&\ell$), requiring all specified labels; gradual union labels ($\ell + \ell$), matching if at least one specified label is present; and empty labels (ε), representing the absence of labels. Following Angles et al. [2023], property types are represented as records consisting of attribute names a and *simple types* τ. Property types can be either open or closed, specifying whether they allow additional properties beyond those explicitly listed. By default, property types are closed, requiring exact attribute definitions. To represent open property types, gradual typing is used by appending the unknown type $\star$ at the end of the record as *gradual rows* ($\{\overline{a_i : \tau_i}^{i}, \star\}$) [Bañados Schwerter et al. 2021]. Intuitively, the unknown type $\star$ denotes zero or more unspecified properties. A bottom property type means errors.

Simple types include basic types such as integer ($\mathbb{Z}$), boolean ($\mathbb{B}$), and string (String), the unknown type ($\star$) matching any simple type, bottom type $\bot$ representing errors, and gradual union types ($\tau + \tau$) [Toro and Tanter 2017], indicating membership in at least one of its subtypes. Importantly, gradual unions differ from traditional unions in their plausibility-based subtyping. For instance, $\mathbb{Z} + \mathbb{B}$ is a subtype of $\mathbb{Z}$ since it is plausible for a value to satisfy this type. While the choice between union types is irrelevant for dynamic semantics, gradual unions are required to under-approximate empty results during typechecking and avoid conservatively raising empty-result warnings for queries such as `(x: a: String + 𝔹 WHERE x.a > 0)`.

Patterns can take several forms. Node patterns ((d)) match graph nodes using descriptors. In a similar fashion, edge patterns ($\overset{d}{\Longleftrightarrow}$) match relationships between nodes. They follow GPC semantics but add one additional directionality from the ISO standard: any direction ($-$). Thus, the supported directions are forward ($\rightarrow$), backward ($\leftarrow$), undirected ($\sim$), and now also any direction ($-$). Union patterns ($\pi + \pi$) match if either subpattern matches. Concatenation patterns ($\pi\ \pi$) combine patterns sequentially, requiring a shared boundary node. Conditioned patterns ($\pi_{\langle t \rangle}$) impose logical constraints on matched patterns. Repeated patterns ($\pi^{l..u}$) match patterns repeated between l and u number of times. *Expressions* (t) define logical conditions on patterns. They include constants c, binary operations (t bop t), unary operations (uop t), projections ($x.a$), casts (t as τ), and type tests (t is τ). Constants include natural numbers (z), booleans (b), and strings (String). Binary operations include addition (+), string concatenation ($\|$), logical conjunction ($\wedge$), logical disjunction ($\vee$), and equality comparisons (=). For simplicity, we consider only negation ($\neg$) as a unary operation. Furthermore, the unknown type $\star$ may be omitted in some cases. For example, $()$ abbreviates $(\_ : \star\ \{\star\})$, and similarly $\Longleftrightarrow$ abbreviates $\overset{\_:\star\ \{\star\}}{\Longleftrightarrow}$. Additionally, the notations (L) and $(\_ : L)$ are considered equivalent, as are $\overset{L}{\Longleftrightarrow}$ and $\overset{\_:L}{\Longleftrightarrow}$. A concatenation pattern $(x_1 : \texttt{Person}\ \{a : \mathbb{Z}\}) \xrightarrow{x_2:\texttt{Student}\ \{a:\mathbb{B}\}} (x_1 : \texttt{Person}\ \{b : \mathbb{B}\})$ describes a sequence where a node labeled Person with an integer attribute a, assigned to variable x_1. This node is followed by a forward edge labeled Student with a boolean attribute a, assigned to variable x_2, which then leads to another node labeled Person with a boolean attribute b, assigned to the same variable x_1.

4.2 Subtyping

The subtyping relations, defined in Figure 3, use the judgment $A \lesssim B$ to denote that A is a subtype of B. We first present descriptor type subtyping, which will be used later to filter nodes and edges during reduction. The subtyping relation for descriptor types $L_1 \lesssim L_2$ ensures that their respective label and record components are also in a subtyping relation.

The subtyping relation for labels ($\ell_1 \lesssim \ell_2$) follows a plausibility-based approach [Garcia et al. 2016]. Two gradual types are considered in a subtyping relation if there exists a pair of fully-precise (concrete) types they represent that satisfy the subtyping rules. Identical labels are subtypes of each other. The unknown label $\star$ is both a subtype and a supertype of any label since it can represent any label. The empty label ε acts as a universal supertype, meaning it encompasses all other labels. Intersection labels $A \& B$ represent entities carrying both labels A and B. Thus, a label ℓ_3 is a supertype of an intersection label $\ell_1 \& \ell_2$ if it is a supertype of either ℓ_1 or ℓ_2. However, because A alone is not a subtype of $A \& B$, the label ℓ_1 is a subtype of $\ell_2 \& \ell_3$ only if it is a subtype of both ℓ_2 and ℓ_3. Gradual unions $\ell_1 + \ell_2$, unlike concrete unions, represent cases where an entity may have either ℓ_1 or ℓ_2. Following an optimistic approach, a gradual union is a subtype of ℓ_3 if at least one of ℓ_1 or ℓ_2 is a subtype of ℓ_3 [Toro and Tanter 2017].

The subtyping of property types ($R_1 \lesssim R_2$) is illustrated at the bottom of Figure 3. The subtyping behavior differs between closed and open records. In closed records $\{\overline{a_i : \tau_i}^{i}\}$, width subtyping is not permitted (rule (Sc)), meaning subtyping holds only if all attributes match and their corresponding simple types are subtypes ($\tau_1 \lesssim \tau_2$). This approach offers greater expressivity, as it allows matching entities with an exact number of properties. A similar concept is introduced in the work of Angles

$\boxed{\ell_1 \text{ is a subtype of } \ell_2\ (\ell_1 \lesssim \ell_2)}$

$$\frac{}{\mathtt{l} \lesssim \mathtt{l}} \qquad \frac{}{\star \lesssim \ell} \qquad \frac{}{\ell \lesssim \star} \qquad \frac{}{\ell \lesssim \varepsilon} \qquad \frac{\ell_1 \lesssim \ell_3}{\ell_1 \& \ell_2 \lesssim \ell_3} \qquad \frac{\ell_2 \lesssim \ell_3}{\ell_1 \& \ell_2 \lesssim \ell_3}$$

$$\frac{\ell_1 \lesssim \ell_3}{\ell_1 + \ell_2 \lesssim \ell_3} \qquad \frac{\ell_2 \lesssim \ell_3}{\ell_1 + \ell_2 \lesssim \ell_3} \qquad \frac{\ell_3 \lesssim \ell_1}{\ell_3 \lesssim \ell_1 + \ell_2} \qquad \frac{\ell_3 \lesssim \ell_2}{\ell_3 \lesssim \ell_1 + \ell_2} \qquad \frac{\ell_1 \lesssim \ell_2 \quad \ell_1 \lesssim \ell_3}{\ell_1 \lesssim \ell_2 \& \ell_3}$$

$\boxed{\tau_1 \text{ is a subtype of } \tau_2\ (\tau_1 \lesssim \tau_2)}$

$$\frac{}{\iota \lesssim \iota} \qquad \frac{}{\bot \lesssim \tau} \qquad \frac{}{\star \lesssim \tau} \qquad \frac{}{\tau \lesssim \star} \qquad \frac{\tau_1 \lesssim \tau_3}{\tau_1 + \tau_2 \lesssim \tau_3}$$

$$\frac{\tau_2 \lesssim \tau_3}{\tau_1 + \tau_2 \lesssim \tau_3} \qquad \frac{\tau_3 \lesssim \tau_1}{\tau_3 \lesssim \tau_1 + \tau_2} \qquad \frac{\tau_3 \lesssim \tau_2}{\tau_3 \lesssim \tau_1 + \tau_2}$$

$\boxed{L_1 \text{ is a subtype of } L_2\ (L_1 \lesssim L_2)}$

$$\frac{\ell_1 \lesssim \ell_2 \quad R_1 \lesssim R_2}{\ell_1\, R_1 \lesssim \ell_2\, R_2}$$

$\boxed{R_1 \text{ is a subtype of } R_2\ (R_1 \lesssim R_2)}$

$$(\mathrm{S}\bot)\,\frac{}{\bot \lesssim R} \qquad (\mathrm{S}c)\,\frac{\forall i. \tau_i \lesssim \tau_i'}{\{\overline{a_i : \tau_i}^i\} \lesssim \{\overline{a_i : \tau_i'}^i\}} \qquad (\mathrm{S}ol)\,\frac{\{\overline{a_i : \tau_i}^i\} \lesssim \{\overline{a_i : \tau_i'}^i\}}{\{\overline{a_i : \tau_i}^i, \star\} \lesssim \{\overline{a_i : \tau_i'}^i, \overline{a_j : \tau_j}^j\}}$$

$$(\mathrm{S}or)\,\frac{\{\overline{a_i : \tau_i}^i\} \lesssim \{\overline{a_i : \tau_i'}^i\}}{\{\overline{a_i : \tau_i}^i, \overline{a_j : \tau_j}^j\} \lesssim \{\overline{a_i : \tau_i'}^i, \star\}} \qquad (\mathrm{S}o)\,\frac{\{\overline{a_i : \tau_i}^i\} \lesssim \{\overline{a_i : \tau_i'}^i\}}{\{\overline{a_i : \tau_i}^i, \overline{a_j : \tau_j}^j, \star\} \lesssim \{\overline{a_i : \tau_i'}^i, \overline{a_k : \tau_k}^k, \star\}}$$

Fig. 3. Consistent Subtyping for Descriptor Types.

et al. [2023]. Furthermore, the ISO has already established the notion of OPEN/CLOSED data types. For open records $\{\overline{a_i : \tau_i}^i, \star\}$, where the unknown type $\star$ can match any attribute, additional fields are allowed on both sides as long as the explicitly defined attributes remain consistent (rules (Sol),(Sor) and (So)). For example, $\{a : \mathbb{Z}, b : \mathbb{B}\}$ is not a subtype of $\{a : \mathbb{Z}\}$ because width subtyping is disallowed for closed records. However, $\{a : \mathbb{Z}, b : \mathbb{B}\}$ is a subtype of $\{a : \mathbb{Z}, \star\}$, and conversely, $\{a : \mathbb{Z}, \star\}$ is a subtype of $\{a : \mathbb{Z}, b : \mathbb{B}\}$. The subtyping relation for simple types follows a structure similar to label types and is illustrated at the middle of Figure 3.

4.3 Reduction

Figure 4 illustrates the dynamic semantics for patterns in FPPC, which largely follows GPC's semantics. Reduction is expressed as $[\![\pi]\!]_G = \{\overline{(p, \mu)}\}$, which denotes that path pattern π reduces to a set of pairs, each consisting of a path p and an assignment μ, under a property graph G. An assignment μ is a partial function mapping a set of variables $\mathcal{X}$ to a set of results $\mathcal{R}$. The results r include node identifiers $\mathcal{N}$, directed and undirected edge identifiers $\mathcal{E}_d \cup \mathcal{E}_u$, null values null, and lists of results $\bar{r}_i$. Note that a pattern can only get stuck if a subexpression t within it is stuck. For illustration purposes, we indicate where expressions can become stuck, but we intentionally avoid propagating stuckness through patterns in order to (1) remain close to the GPC semantics and (2) avoid adding complexity to an already exotic semantic model.

For a *node pattern* $(x? : L)$, each node in the graph is checked to determine whether its descriptor type L' satisfies the descriptor L. Unlike GPC, where only labels are used for matching, FPPC supports additional features such as gradual labels, intersection and union labels, and property types. Consequently, verifying whether a node matches a descriptor type is more involved and is done by checking if L' is a *subtype* of L.

$$\boxed{[\![\pi]\!]_G = \{\overline{(p,\mu)}\}}$$

$$[\![(x? : L)]\!]_G = \left\{(n, x?\widetilde{\mapsto} n) \;\middle|\; n \in N_{id}, \vdash \mathcal{L}(n)\ \mathcal{P}(n) : L', L' \lesssim L\right\} \qquad (\text{R}())$$

$$[\![\xrightarrow{x?:L}]\!]_G = \left\{(n_1 e n_2, x?\widetilde{\mapsto} e) \;\middle|\; e \in E_d, n_1 = \mathsf{src}(e), n_2 = \mathsf{tgt}(e), \vdash \mathcal{L}(e)\ \mathcal{P}(e) : L', L' \lesssim L\right\} \qquad (\text{R}\longrightarrow)$$

$$[\![\xleftarrow{x?:L}]\!]_G = \left\{(n_2 e n_1, x?\widetilde{\mapsto} e) \;\middle|\; e \in E_d, n_1 = \mathsf{src}(e), n_2 = \mathsf{tgt}(e), \vdash \mathcal{L}(e)\ \mathcal{P}(e) : L', L' \lesssim L\right\} \qquad (\text{R}\longleftarrow)$$

$$[\![\overset{x? : L}{\sim\!\sim}]\!]_G = \left\{(n_1 e n_2, x?\widetilde{\mapsto} e) \;\middle|\; e \in E_u, \mathsf{endpoints}(e) = \{n_1, n_2\}, \vdash \mathcal{L}(e)\ \mathcal{P}(e) : L', L' \lesssim L\right\} \qquad (\text{R}\sim)$$

$$[\![\overset{x?:L}{\text{---}}]\!]_G = [\![\overset{x? : L}{\sim\!\sim}]\!]_G \cup [\![\xrightarrow{x?:L}]\!]_G \cup [\![\xleftarrow{x?:L}]\!]_G \qquad (\text{R}-)$$

$$[\![\pi_1\ \pi_2]\!]_G = \left\{(p_1 \cdot p_2, \mu_1 \cdot \mu_2) \;\middle|\; (p_i, \mu_i) \in [\![\pi_i]\!]_G, p_1 \text{ and } p_2 \text{ concatenate}, \mu_1 \text{ and } \mu_2 \text{ concatenate}\right\} \qquad (\text{R}\cdot)$$

$$[\![\pi_1 + \pi_2]\!]_G = \left\{(p, \mathsf{fillNulls}(\mu, \mathsf{freevar}(\pi_1 + \pi_2))) \;\middle|\; (p, \mu) \in [\![\pi_1]\!]_G \cup [\![\pi_2]\!]_G\right\} \qquad (\text{R}+)$$

$$[\![\pi_{\langle t \rangle}]\!]_G = \left\{(p, \mu) \in [\![\pi]\!]_G \;\middle|\; [\![t]\!]_{\mu,\delta} = \mathsf{true}\right\} \qquad (\text{R}\theta)$$

$$[\![\pi^{l..u}]\!]_G = \bigcup_{i=l}^{u} [\![\pi]\!]_G^i \qquad (\text{R}l..u)$$

Fig. 4. Dynamic Semantics for Patterns.

The descriptor type L' of a node is determined by two functions: the labeling function $\mathcal{L}(n)$, which expresses label sets as a conjunction of labels, and the property function $\mathcal{P}(n)$, which represents partial functions as a record mapping attribute names to constant values. The inference rule for computing the descriptor type is:

$$\frac{}{\vdash \ell\ \{\overline{a_i = c_i}^{\,i}\} : \ell\ \{\overline{a_i : ty(c_i)}^{\,i}\}}$$

where $ty(\cdot)$ is a function that returns the simple type of a constant. Rule (R()) returns a node path n for each node that satisfies the descriptor and an optional mapping $\widetilde{\mapsto}$ from the variable $x?$ to the selected node n. Since variables can be optional, if no variable is present (_), the mapping remains empty; otherwise, the variable is assigned to the selected node. For example, the node pattern $(x$: `Person` {`name` : $\star$, `status` : `String`}$)$ evaluates to $\{(n_2, x \mapsto n_2)\}$. Here, nodes n_1 and n_2 are initially selected because they match label `Person`. However, only n_1 satisfies the property condition, as it contains attribute `status` with a string value.

The reduction of edge patterns follows a similar approach to node patterns, but now directionality must be considered. For directed edges $\xrightarrow{x:L}$ and $\xleftarrow{x:L}$, both the source and target nodes must be checked to ensure they align with the specified direction. In contrast, undirected edges return only the two endpoints of the matching edges. The result set of an any-direction edge pattern is computed as the union of the result sets for forward, backward, and undirected edges.

Rule (R·) denotes the reduction of *concatenation patterns* $\pi_1\ \pi_2$, where the resulting path and assignment are obtained by concatenating the results of π_1 and π_2. Two paths, p_1 and p_2 concatenate, if the last node of p_1 is equal to the first node of p_2. The new path, $p_1 \cdot p_2$, is formed by connecting the two path sequences, e.g. $n_3e_3n_1 \cdot n_1e_1n_2 = n_3e_3n_1e_1n_2$. Two assignments, μ_1 and μ_2 concatenate, if for all $x \in dom(\mu_1) \cap dom(\mu_2)$, it holds that $\mu_1(x) = \mu_2(x)$. The concatenation operation $\mu_1 \cdot \mu_2$ is defined as follows: for each variable $x \in dom(\mu_1) \cup dom(\mu_2)$, if $x \in dom(\mu_1)$, then $(\mu_1 \cdot \mu_2)(x) = \mu_1(x)$; otherwise, $(\mu_1 \cdot \mu_2)(x) = \mu_2(x)$. For example, concatenating the two patterns $(x$: `Person` {`name` : $\star$, `status` : `String`}$)$ $\xleftarrow{y:\texttt{Likes}\ \{\star\}}$ reduces to empty $\{\}$. Since n_2 can not be concatenated with $n_3e_2n_2$.

Rule (R+) applies to *union patterns* $\pi_1 + \pi_2$, where the resulting path is drawn from either π_1 or π_2. The resulting assignment is computed using the fillNulls function, defined as follows: for each

$$[\![c]\!]_{\mu,\delta} = \textbf{ok}\ c \tag{Rc}$$

$$[\![x.a]\!]_{\mu,\delta} = \begin{cases} \textbf{stuck} & \text{if } x \notin dom(\mu) \\ \textbf{ok}\ \delta(\mu(x))(a) & \text{if } a \in dom(\delta(\mu(x))) \\ \textbf{ok}\ \mathsf{null} & \text{otherwise} \end{cases} \tag{Ra}$$

$$[\![t \mathrel{\mathsf{is}} \tau]\!]_{\mu,\delta} = \mathsf{for}\ (v \leftarrow [\![t]\!]_{\mu,\delta})\ \mathsf{yield}\ (ty(v) \lesssim \tau) \tag{Ris}$$

$$[\![t \mathrel{\mathsf{as}} \tau]\!]_{\mu,\delta} = \mathsf{for}\ (v \leftarrow [\![t]\!]_{\mu,\delta})\ \mathsf{yield}\ (v \Rightarrow \tau) \tag{Ras}$$

$$[\![t_1 \mathrel{\mathsf{bop}} t_2]\!]_{\mu,\delta} = \begin{cases} \textbf{stuck} \quad \text{if } \triangle(\mathsf{bop}) \text{ is not defined or } [\![\mathsf{bop}]\!] \text{ is not defined} \\ \mathsf{for}\ (v_1 \leftarrow [\![t_1 \mathrel{\mathsf{as}} \tau_1]\!]_{\mu,\delta};\ v_2 \leftarrow [\![t_2 \mathrel{\mathsf{as}} \tau_2]\!]_{\mu,\delta}) \\ \qquad\qquad \mathsf{yield}\ (v_1\ [\![\mathsf{bop}]\!]\ v_2) \quad \text{if } \triangle(\mathsf{bop}) = \tau_1 \times \tau_2 \rightarrow \tau_3 \end{cases} \tag{Rbop}$$

$$[\![\mathsf{uop}\ t]\!]_{\mu,\delta} = \begin{cases} \textbf{stuck} \quad \text{if } \triangle(\mathsf{uop}) \text{ is not defined or } [\![\mathsf{uop}]\!] \text{ is not defined} \\ \mathsf{for}\ (v \leftarrow [\![t \mathrel{\mathsf{as}} \tau_1]\!]_{\mu,\delta})\ \mathsf{yield}\ ([\![\mathsf{uop}]\!]\ v) \quad \text{if } \triangle(\mathsf{uop}) = \tau_1 \rightarrow \tau_2 \end{cases} \tag{Ruop}$$

Fig. 5. Dynamic Semantics for Expressions.

variable $x \in \overline{x}$, if $x \in dom(\mu)$, then $\mathsf{fillNulls}(\mu, \overline{x})(x) = \mu(x)$; otherwise, $\mathsf{fillNulls}(\mu, \overline{x})(x) = \mathsf{null}$. In other words, the resulting assignment maps any free variable in $\mathsf{freevar}(\pi_1 + \pi_2)$ but not in $dom(\mu)$ to null. For instance, the pattern $(x : \mathtt{Person}\ \{\mathtt{name} : \star, \mathtt{status} : \mathtt{String}\}) + (y : \mathtt{Person}\ \{\mathtt{name} : \star, \mathtt{status} : \mathbb{Z}\})$ reduces the pattern $(x : \mathtt{Person}\ \{\mathtt{name} : \star, \mathtt{status} : \mathtt{String}\})$ to $\{(n_1, x \mapsto n_1)\}$, and $(y : \mathtt{Person}\ \{\mathtt{name} : \star, \mathtt{status} : \mathbb{Z}\})$ to $\{(n_2, y \mapsto n_2)\}$. Then the final result is $\{(n_1, [x \mapsto n_1, y \mapsto \mathsf{null}]), (n_2, [y \mapsto n_2, x \mapsto \mathsf{null}])\}$. Rule R$\theta$ applies to *conditioned patterns* $\pi_{\langle t \rangle}$, where the result set of π is returned only if the conditioned expression t evaluates to true.

Repetitions. Rule R$l..u$ defines the reduction of *repetition patterns* $\pi^{l..u}$, which generate the union of results from l to u repetitions of π. The base case for zero repetitions is:

$$[\![\pi]\!]^0_G = \{(n, \mu) \mid n \text{ is a node in } G\}$$

where μ maps every variable in $\mathsf{freevar}(\pi)$ to the empty list $[\,]^l$. [5] Intuitively, this makes sense since the set of all nodes is concatenable with any other set of nodes. For $i > 0$, the case is:

$$[\![\pi]\!]^i_G = \left\{(p, \mu) \,\middle|\, (p_1, \mu_1), \ldots, (p_i, \mu_i) \in [\![\pi]\!]_G, p = p_1 \cdot \ldots \cdot p_i, \mu = \mathsf{collect}(\mu_1, \ldots, \mu_i)\right\}$$

where $\forall x \in \mathsf{freevar}(\pi).\mathsf{collect}(\mu_1, \ldots, \mu_i)(x) = [\mu_1(x), \ldots, \mu_i(x)]^l$. To avoid unnecessary complexity in the model, we assume that all paths p_j have a positive length, where length is computed as the number of edges identifiers in a path. That is, they contain at least one edge. As discussed in GPC, allowing zero-length paths could lead to infinite result sets. For instance, pattern $\overset{x : \mathsf{Knows}\ \{\star\}}{\smile}$ reduces to $\{(n_1 e_1 n_2, x \mapsto e_1), (n_2 e_1 n_1, x \mapsto e_1)\}$. Therefore, $\overset{x : \mathsf{Knows}\ \{\star\}}{\smile}{}^{1..2}$ reduce as follows: $[\![\overset{x : \mathsf{Knows}\ \{\star\}}{\smile}]\!]^1_G$ evaluates to $\{(n_1 e_1 n_2, x \mapsto [e_1]^1), (n_2 e_1 n_1, x \mapsto [e_1]^1)\}$, and $[\![\overset{x : \mathsf{Knows}\ \{\star\}}{\smile}]\!]^2_G$ to $\{(n_1 e_1 n_2 e_1 n_1, x \mapsto [e_1, e_1]^1), (n_2 e_1 n_1 e_1 n_2, x \mapsto [e_1, e_1]^1)\}$. The resulting set is the union of two sets.

Expression reduction. Expression reduction is presented in Figure 5. Expressions reduce under an assignment μ and partial function δ. Expressions evaluate to values, which include constants (c), nulls (null), or errors (**error**). For illustration purposes, we include a stuck state (**stuck**) and

[5] Contrary to GPC, we index lists of results using l from the repetition query $\pi^{l..u}$. This indexing is required solely for type safety. Accordingly, unification $\mu_1 \cdot \mu_2$ is defined by equating lists while ignoring such indices, and producing a list with the maximum of the two repetition bounds.

Node Types	$N ::= (\!\| L \|\!)$	Variable Types	$T ::= N \mid E \mid T + T \mid [T]^l \mid \bot \mid \mathsf{Null}$
Edge Types	$E ::= N \xrightarrow{L} N \mid N \overset{L}{\sim} N$	Type Environments	$\Gamma ::= \cdot \mid \Gamma, x \mapsto T$
Path Types	$P ::= N \mid P - N \mid P + P \mid \bot$	Schemas	$S ::= \cdot \mid S, E \mid S, N$

Fig. 6. Syntax of Types.

explicitly mark cases where evaluation may get stuck in grey. Although this is not part of the formal semantics, it helps illustrate the kinds of errors that the type system is designed to prevent. To avoid explicit propagation of errors (and stuck states), expression reduction follows a *monadic style*, representing computations that may fail with "optional" values that can either succeed (tagged as **ok**) or result in an **error** (or stuck state). Rule (Ra) produces a stuck state if the variable x is not in the domain of the assignment μ; otherwise it checks whether the attribute a is present in the entity associated to x in the current assignment, and returns the corresponding value; otherwise, instead of raising an error, it returns null. This makes the semantics less conservative as expression true $\vee$ null yields true under a three-value-logic semantics instead of an error. A type test expression t is τ first evaluates t and then checks whether the type of the value is a subtype of τ (rule Ris). In contrast, cast expressions t as τ enforce type conversion, directly transforming a value to type τ as the result. If casting is impossible (e.g., converting the string 'a' to $\mathbb{Z}$), a type error (**error**) is raised. Note that type errors do not propagate to path patterns; they yield empty outputs since **error** is not equivalent to true. On the other hand stuck states terminate the evaluation of the top-level query.

5 Type System

This section presents the type system of FPPC, highlighting key differences and improvements over the work of Francis et al. [2023a]. The main enhancement is the adding schemas as contexts, which allows for more precise verification of pattern validity. By considering the schema, the system can rule out paths that do not match entities in the database. The type system produces two outputs: the path type and the mappings of variables to their respective pattern types. These outputs help detect errors, such as references to non-existent variables and paths that do not return results. Moreover, our type system is less strict in some cases.

5.1 Syntax

Figure 6 presents the syntax of types, drawing inspiration from PG-Schema [Angles et al. 2023] and the ISO standard [ISO/IEC 2024]. *Node types* are simply indexed by a descriptor type L (Fig. 2). Meanwhile, *Edge types*, representing the types of edges, include the type of the nodes they connect as well as a descriptor type. Only two edge directions are considered: directed (denoted by $\longrightarrow$) and undirected (denoted by $\sim$), instead of four, since any backward edge can be expressed as a forward edge, and the "any" direction can be captured using union types.

Path types (P) specify the first and last node type of a path and encode relevant intermediate node type information. They are used to determine whether two patterns can be concatenated and to predict cases where a path pattern would return an empty result. Path types support union types ($P + P$) and include the bottom type ($\bot$), representing an empty or unreachable path.

Variable types (T) assign types to nodes and edges captured by variables. They include node types for node patterns, edge types for edge patterns, union types ($T + T$) for union patterns, and list types ($[T]^l$) for repetition patterns, where each list is indexed by its minimal cardinality to help predict empty results. They also include the bottom type ($\bot$), indicating type errors, and the null type (Null), which explicitly represents the absence of a variable in a subquery.

Finally, the *type environment* (Γ) maps variables to their corresponding variable types, and a *schema* (S) consists of a (possibly empty) list of edge types and node types.

5.2 Subtyping

Subtyping plays a key role in the typing rules of FPPC, by enabling the system to filter nodes and edges from the schema that match a given type. The subtyping relation for type variables, denoted $T_1 \lesssim T_2$, is defined inductively on the structure of types:

$$\frac{L_1 \lesssim L_2}{(\!|L_1|\!) \lesssim (\!|L_2|\!)} \qquad \frac{L_1 \lesssim L_2 \quad N_1 \lesssim N_3 \quad N_2 \lesssim N_4}{N_1 \xrightarrow{L_1} N_2 \lesssim N_3 \xrightarrow{L_2} N_4} \qquad \frac{L_1 \lesssim L_2 \quad N_1 \lesssim N_3 \quad N_2 \lesssim N_4}{N_1 \overset{L_1}{\sim} N_2 \lesssim N_3 \overset{L_2}{\sim} N_4}$$

$$\frac{L_1 \lesssim L_2 \quad N_1 \lesssim N_4 \quad N_2 \lesssim N_3}{N_1 \overset{L_1}{\sim} N_2 \lesssim N_3 \overset{L_2}{\sim} N_4} \qquad \frac{T_1 \lesssim T_2}{[T_1]^l \lesssim [T_2]^l} \qquad \frac{T_1 \lesssim T}{T_1 + T_2 \lesssim T}$$

$$\frac{T_2 \lesssim T}{T_1 + T_2 \lesssim T} \qquad \frac{T \lesssim T_1}{T \lesssim T_1 + T_2} \qquad \frac{T \lesssim T_2}{T \lesssim T_1 + T_2} \qquad \frac{}{\bot \lesssim T} \qquad \frac{}{\mathsf{Null} \lesssim \mathsf{Null}}$$

In particular, subtyping for type descriptors is covered in §4.2, and, similar to labels, a union type $T_1 + T_2$ is subtype of T if either T_1 or T_2 is a subtype of T.

5.3 Meet Operator

The meet operation $\sqcap$ determines the most precise type information when comparing two types.

For *label types*, if either label is the unknown label, the result is the other label. Otherwise, the result is their intersection, meaning $\ell_1 \sqcap \ell_2 = \ell_1 \& \ell_2$. Intuitively, if an entity is known to have both labels ℓ_1 and ℓ_2, then it must have the conjunction of these labels, $\ell_1 \& \ell_2$.

For *simple types*, the meet operation is defined as follows, returning $\bot$ for undefined cases:

$$\frac{}{\tau \sqcap \tau = \tau} \qquad \frac{}{\tau \sqcap \star = \tau} \qquad \frac{}{\star \sqcap \tau = \tau} \qquad \frac{\tau_1 \sqcap \tau = \tau_3 \quad \tau_2 \sqcap \tau = \tau_4}{\tau_1 + \tau_2 \sqcap \tau = \tau \sqcap \tau_1 + \tau_2 = \tau_3 \sqcup \tau_4}$$

For simple union types, the meet operation is applied component-wise, and the results are combined using the *join* operator ($\sqcup$). This operator, also used for variable and path types, returns the other element if one is $\bot$, and otherwise returns their union ($\cdot + \cdot$).

For *property types*, the meet operation is defined inductively by applying it to common attributes and, when one operand is an open property type, extending the record with attributes present only in the other. This ensures that the resulting type retains the most precise attribute information while allowing flexibility when merging open and closed property types. The meet returns $\bot$ for undefined cases.

$$\frac{}{\{\overline{a_i \mapsto \tau_{i1}}^{i}\} \sqcap \{\overline{a_i \mapsto \tau_{i2}}^{i}\} = \{\overline{a_i \mapsto \tau_{i1} \sqcap \tau_{i2}}^{i}\}}$$

$$\frac{}{\{\overline{a_i \mapsto \tau_{i1}}^{i}, \overline{a_j \mapsto \tau_j}^{j}, \star\} \sqcap \{\overline{a_i \mapsto \tau_{i2}}^{i}, \overline{a_k \mapsto \tau_k}^{k}, \star\} = \{\overline{a_i \mapsto \tau_{i1} \sqcap \tau_{i2}}^{i}, \overline{a_j \mapsto \tau_j}^{j}, \overline{a_k \mapsto \tau_k}^{k}, \star\}}$$

$$\frac{}{\{\overline{a_i \mapsto \tau_{i1}}^{i}, \star\} \sqcap \{\overline{a_i \mapsto \tau_{i2}}^{i}, \overline{a_k \mapsto \tau_k}^{k}\} = \{\overline{a_i \mapsto \tau_{i1} \sqcap \tau_{i2}}^{i}, \overline{a_k \mapsto \tau_k}^{k}\}}$$

$$\frac{}{\{\overline{a_i \mapsto \tau_{i1}}^{i}, \overline{a_j \mapsto \tau_j}^{j}\} \sqcap \{\overline{a_i \mapsto \tau_{i2}}^{i}, \star\} = \{\overline{a_i \mapsto \tau_{i1} \sqcap \tau_{i2}}^{i}, \overline{a_j \mapsto \tau_j}^{j}\}}$$

The meet of two *descriptor types* is obtained by computing the meet of their respective labels and property types: $\ell_1\ R_1 \sqcap \ell_2\ R_2 = (\ell_1 \sqcap \ell_2)\ (R_1 \sqcap R_2)$.

For *variable types*, the meet is defined inductively following the structure of the types.

$$\frac{}{(\!|L_1|\!) \sqcap (\!|L_2|\!) = (\!|L_1 \sqcap L_2|\!)} \qquad \frac{}{N_{11} \xrightarrow{L_1} N_{12} \sqcap N_{21} \xrightarrow{L_2} N_{22} = (N_{11} \sqcap N_{21}) \xrightarrow{L_1 \sqcap L_2} (N_{21} \sqcap N_{22})}$$

$$\frac{N_{11} \overset{L_1}{\sim} N_{12} \sqcap N_{21} \overset{L_2}{\sim} N_{22} = N_{31} \overset{L_3}{\sim} N_{32} \qquad N_{11} \overset{L_1}{\sim} N_{12} \sqcap N_{22} \overset{L_2}{\sim} N_{21} = N_{41} \overset{L_4}{\sim} N_{42}}{N_{11} \overset{L_1}{\sim} N_{12} \sqcap N_{21} \overset{L_2}{\sim} N_{22} = N_{31} \overset{L_3}{\sim} N_{32} \sqcup N_{41} \overset{L_4}{\sim} N_{42}}$$

$$\frac{T \sqcap T_2 = T_3 \qquad T \sqcap T_1 = T_4}{T \sqcap (T_1 + T_2) = (T_1 + T_2) \sqcap T = T_3 \sqcup T_4} \qquad \frac{T_1 \sqcap T_2 = T_3}{[T_1]^{l_1} \sqcap [T_2]^{l_2} = [T_3]^{\max(l_1, l_2)}}$$

The only notable cases in the meet operation involve undirected edges, unions, and lists. For undirected edges, since there is no inherent direction between nodes, both possible orientations are considered, and the results are then joined together. For union types, the meet is computed by applying the meet operation to each element of the union individually and then joining the results. In the case of list types, the meet takes the maximum of the two repetition bounds as the new index. The intuition is that the intersection (unification) of lists with at least l_1 elements and at least l_2 elements should yield a list with at least $\max(l_1, l_2)$ elements.

5.4 Refinement Operator

Armed with the meet operator, we now define the refinement operation under a schema S to extract possible types for a variable type, a path pattern, a type environment, a path type, or a path type associated to a repetition.

The judgment $S \vdash T_1 \triangleright T_2$ denotes that variable type T_1 is refined to a (potentially) more precise variable type T_2. This refinement is defined inductively as follows:

$$\frac{}{S \vdash N \triangleright \bot \sqcup \bigsqcup_{N' \lesssim N \in S} N \sqcap N'} \qquad \frac{}{S \vdash E \triangleright \bot \sqcup \bigsqcup_{E' \lesssim E \in S} E \sqcap E'}$$

$$\frac{S \vdash T_1 \triangleright T_1' \qquad S \vdash T_2 \triangleright T_2'}{S \vdash T_1 + T_2 \triangleright T_1' \sqcup T_2'} \qquad \frac{S \vdash T \triangleright T'}{S \vdash [T]^l \triangleright [T']^l} \qquad \frac{}{S \vdash \bot \triangleright \bot}$$

The most interesting cases involve node and edge type refinement, both handled by applying the meet operation to each subtype node or edge in the schema and joining the results to produce the refined type. If no matching node or edge type exists, meaning that no type in the schema satisfies the input type, the bottom type ($\bot$) is returned, indicating that the type is unsatisfiable.

The judgment $S \vdash \pi \triangleright T$ states that a node or edge pattern π refines to a variable type T. For space reasons, we illustrate only the most relevant cases.

$$\frac{S \vdash (\!|L|\!) \triangleright T}{S \vdash (x? : L) \triangleright T} \qquad \frac{S \vdash \xrightarrow{L} \triangleright T}{S \vdash \xrightarrow{x?:L} \triangleright T} \qquad \frac{S \vdash \xrightarrow{L} \triangleright T}{S \vdash \xleftarrow{x?:L} \triangleright T} \qquad \frac{S \vdash \xrightarrow{x?:L} \triangleright T_1 \qquad S \vdash \overset{x?:L}{\sim} \triangleright T_2}{S \vdash \underline{x?:L} \triangleright T_1 \sqcup T_2}$$

For edge patterns in the backward direction, the definition is the same as for forward direction refinement, since types are defined in only one direction. The distinction between directions becomes relevant during type checking, as explained §5.5. For edge patterns with any direction, the refinement is defined as the join of the refinements for the forward and undirected cases.

Refinement and the meet operation for type environments are defined together to avoid refining types that do not gain precision, meaning those that are not involved in the meet operation, since they were already refined at some point. The judgment $S \vdash \Gamma_1 \sqcap \Gamma_2 \triangleright \Gamma$ denotes that the meet between type environments Γ_1 and Γ_2 refines to Γ, and it is formally defined as follows:

$$\frac{T_{i1} \sqcap T_{i2} = T_i \qquad S \vdash T_i \triangleright T_i'}{S \vdash (\overline{x_i \mapsto T_{i1}}^{i}, \overline{x_j \mapsto T_j}^{j}) \sqcap (\overline{x_i \mapsto T_{i2}}^{i}, \overline{x_k \mapsto T_k}^{k}) \triangleright \overline{x_i \mapsto T_i'}^{i}, \overline{x_j \mapsto T_j}^{j}, \overline{x_k \mapsto T_k}^{k}}$$

Note that only types associated with common variables are refined, while other variables are directly carried over to the resulting type environment without modification.

Similarly, meet and refinement for path types are defined together. The judgment $S \vdash P_1 \sqcap P_2 \triangleright P_3$ states that the meet between path P_1 and P_2 refines to path type P_3, and it is defined as follows:

$$\frac{S \vdash (\!|L_1 \sqcap L_2|\!) \triangleright (\!|L_3|\!)}{S \vdash (\!|L_1|\!) \sqcap (\!|L_2|\!) \triangleright (\!|L_3|\!)} \qquad \frac{S \vdash N_2 \sqcap N_3 \triangleright N_4}{S \vdash (P_1 - N_2) \sqcap N_3 \triangleright P_1 - N_4} \qquad \frac{S \vdash P_1 \sqcap P_2 \triangleright P_3}{S \vdash P_1 \sqcap (P_2 - N_3) \triangleright P_3 - N_3}$$

$$\frac{S \vdash P_1 \sqcap P_2 \triangleright P_4 \qquad S \vdash P_1 \sqcap P_3 \triangleright P_5}{S \vdash P_1 \sqcap (P_2 + P_3) \triangleright P_4 \sqcup P_5} \qquad \frac{S \vdash P_2 \sqcap P_1 \triangleright P_4 \qquad S \vdash P_3 \sqcap P_1 \triangleright P_5}{S \vdash (P_2 + P_3) \sqcap P_1 \triangleright P_4 \sqcup P_5}$$

For two single node path types, the result is obtained by applying the meet operator to their descriptor types. However, if either P_1 or P_2 consists of multiple nodes, the meet operation is applied only to the last node type of P_1 and the first node type of P_2. This ensures that only the boundary nodes are refined while preserving the internal structure of each path type.

Finally, the judgment $S \vdash P_1^l \triangleright P_2$ denotes that the repetition of a path P_1 at least l times (where $l \geq 0$) refines to P_2. Validation of a path repetition is performed by computing its self-meet l times. The repetition and refinement process is defined inductively as follows.

$$\frac{S \vdash P^l \triangleright P_1 \qquad S \vdash P_1 \sqcap P \triangleright P_2}{S \vdash P^{l+1} \triangleright P_2} \qquad \frac{}{S \vdash P^0 \triangleright (\!|\star|\!)}$$

Remember that in the dynamic semantics (§4.3), zero repetitions correspond to every node, preventing unnecessary failing joins. Similarly, at the type level, zero repetitions of a path type result in the fully imprecise node type, which matches all node types. This ensures that concatenating a path with zero repetitions with another path never results in an error ($\bot$).

5.5 Typing

Figure 7 presents the typing rules of FPPC. The judgment $S \vdash \pi : P;\ \Gamma$ denotes that pattern π has path type P and type environment Γ, under schema S. The type system of GPC primarily determines whether a variable represents a node or an edge. While this ensures that variables do not conflict, it provides only a coarse level of validation. Additionally, the typing rules in GPC are verbose, requiring multiple cases for concatenation patterns and five separate rules for union patterns. The type system of FPPC is more compact and goes beyond simple variable conflict checks. It (1) verifies whether the pattern is satisfiable within the schema, (2) ensures type consistency between concatenated patterns, (3) refine types by combining both pattern and schema type information, (4) under-approximates empty results, and (5) over-approximates program stuck states.

Node and Edge patterns. Rules (Tnode) and (Tedge) type-check node and edge patterns using a similar process. The path type and variable type are obtained via the refinement operation, described in the previous section, which searches the schema for possible matches. As a result, the typing environment is a (possibly empty) mapping from variables to their corresponding variable types (T). However, to derive a path type, we use the function $\lfloor \cdot \rfloor^{\Leftrightarrow}$, defined in the Appendix. This metafunction is indexed by a direction $\Leftrightarrow$, which represents the desired path type direction, relevant only when typing edges. For node types and the bottom type, $\lfloor \cdot \rfloor^{\Leftrightarrow}$ returns the type itself, as

$\boxed{S \vdash \pi : P;\ \Gamma}$

$$\text{(Tnode)}\ \frac{S \vdash (x? : L) \triangleright T}{S \vdash (x? : L) : \lfloor T \rfloor^{-};\ x?\ \widetilde{\mapsto}\ T} \qquad \text{(Tedge)}\ \frac{S \vdash \xLeftrightarrow{x?:L} \triangleright T}{S \vdash \xLeftrightarrow{x?:L} : \lfloor T \rfloor^{\Leftrightarrow};\ x?\ \widetilde{\mapsto}\ T}$$

$$\text{(Tcat)}\ \frac{\begin{array}{cc} S \vdash \pi_1 : P_1;\ \Gamma_1 & S \vdash \pi_2 : P_2;\ \Gamma_2 \\ S \vdash P_1 \sqcap P_2 \ \triangleright\ P & S \vdash \Gamma_1 \sqcap \Gamma_2 \triangleright \Gamma \end{array}}{S \vdash \pi_1\ \pi_2 : P;\ \Gamma} \qquad \text{(Tu)}\ \frac{S \vdash \pi_1 : P_1;\ \Gamma_1 \qquad S \vdash \pi_2 : P_2;\ \Gamma_2}{S \vdash \pi_1 + \pi_2 : P_1 \sqcup P_2;\ (\Gamma_1 \sqcup \Gamma_2)}$$

$$\text{(Tcon)}\ \frac{S \vdash \pi : P;\ \Gamma \qquad \Gamma \vdash t : \tau \qquad \tau \sqcap \mathbb{B} \neq \bot}{S \vdash \pi_{\langle t \rangle} : P;\ \Gamma} \qquad \text{(Tfail)}\ \frac{S \vdash \pi : P;\ \Gamma \qquad \Gamma \vdash t : \tau \qquad \tau \sqcap \mathbb{B} = \bot}{S \vdash \pi_{\langle t \rangle} : \bot;\ \Gamma}$$

$$\text{(Trep)}\ \frac{S \vdash \pi : P;\ \Gamma \qquad len(P) > 0 \qquad 0 \leq l \leq u \qquad S \vdash P^{\min(l,2)} \triangleright P'}{S \vdash \pi^{l..u} : P';\ [\Gamma]^{l}}$$

$\boxed{\Gamma \vdash t : \tau}$

$$\text{(Tc)}\ \frac{}{\Gamma \vdash c : ty(c)} \qquad \text{(Tpj)}\ \frac{x \in dom(\Gamma)}{\Gamma \vdash x.a : \Gamma(x)(a)} \qquad \text{(Tas)}\ \frac{\Gamma \vdash t : \tau'}{\Gamma \vdash t \ \mathsf{as}\ \tau : \tau \sqcap \tau'} \qquad \text{(Tis)}\ \frac{\Gamma \vdash t : \tau'}{\Gamma \vdash t \ \mathsf{is}\ \tau : \mathbb{B}}$$

$$\text{(Tbop)}\ \frac{\triangle(\mathsf{bop}) = \tau_1 \times \tau_2 \rightarrow \tau_3 \qquad \Gamma \vdash t_1 : \tau_1' \qquad \Gamma \vdash t_2 : \tau_2'}{\Gamma \vdash t_1\ \mathsf{bop}\ t_2 : \widetilde{cod}(\tau_1 \sqcap \tau_1' \times \tau_2 \sqcap \tau_2' \rightarrow \tau_3)} \qquad \text{(Tuop)}\ \frac{\triangle(\mathsf{uop}) = \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash t : \tau_1'}{\Gamma \vdash \mathsf{uop}\ t : \widetilde{cod}(\tau_1 \sqcap \tau_1' \rightarrow \tau_2)}$$

Fig. 7. Type System for FPPC.

direction is not relevant. For edges, the resulting path type depends on the specified direction. If the required direction is forward (⟶), the node types maintain their order. If the direction is backward (⟵), the order is reversed. If the direction is any (–) or the edge is undirected, the resulting path type includes both directions using union types, as both kinds of paths may be returned.

Concatenation patterns. In the dynamic semantics (§4.3), two patterns are concatenated by connecting paths while ensuring that the boundary nodes match and unifying assignments, requiring that common variables map to the same values. Rule (Tcat) follows a similar approach at the type level. First, the path types P_1 and P_2, along with the typing environments Γ_1 and Γ_2, are derived recursively for P_1 and P_2, respectively. Then the resulting path type and type environment is computed using their corresponding refinement operators.

Union patterns. A union pattern $\pi_1 + \pi_2$ represents a path that follows either π_1 or π_2. In the dynamic semantics, the resulting assignment is computed by unifying common variables from both subpatterns and assigning null values to variables that appear in only one of them. Rule (Tu) type-checks both subpatterns π_1 and π_2, deriving their corresponding path types P_1 and P_2, as well as their type environments Γ_1 and Γ_2. The resulting path type and type environment are then computed by taking the join of the corresponding types. Similarly, the type environment is obtained as the join of both environments, Γ_1 and Γ_2. The resulting type environment is computed as the join of both environments defined as follows:

$$\frac{\Gamma_1 = \overline{x_i \mapsto T_{i1}}^{i}, \overline{x_j \mapsto T_j}^{j} \qquad \Gamma_2 = \overline{x_i \mapsto T_{i2}}^{i}, \overline{x_k \mapsto T_k}^{k}}{\Gamma_1 \sqcup \Gamma_2 = \overline{x_i \mapsto T_{i1} \sqcup T_{i2}}^{i}, \overline{x_j \mapsto T_j + \mathsf{Null}}^{j}, \overline{x_k \mapsto T_k + \mathsf{Null}}^{k}}$$

For common variables, their variable types are joined together. For variables present in only one subpattern, the resulting variable type is the union of the type with the Null type, akin to a Maybe type in GPC. Unlike concatenation patterns, where consistency checking ensures that variable types match, union patterns use type unions. This distinction is necessary, as demonstrated by the pattern $(x : \star\ \{a : \mathbb{Z}\}) + (x : \star\ \{b : \mathbb{B}\})$, which allows matching nodes that have either property a or b, but not necessarily both. Similarly, the type system accepts patterns like $(x : \star\ \{\star\})+ \overset{x:\star\ \{\star\}}{\Longleftrightarrow}$, which are rejected by GPC. However, no inherent semantic issue is observed; from the perspective of the semantic rules, there is nothing fundamentally incorrect about allowing such patterns.

Conditioned patterns. Rules (Tcon) and (Tfail) apply to conditioned patterns of the form $\pi_{\langle t \rangle}$. The pattern π is first type-checked to determine its path type P and type environment Γ. The type environment is then used as input to type-check the condition expression t, which is assigned a type τ, as shown in Fig. 7. To ensure that t is a boolean condition, the meet $\tau_1 \sqcap \tau_2$ is computed. If the result is $\bot$, rule (Tfail) applies, assigning the bottom path type $\bot$ and preserving the type environment Γ. Otherwise, rule (Tcon) returns the original path type P and type environment Γ.

Repetition Patterns. A repetition pattern $\pi^{l..u}$ is type-checked using rule (Trep). To prevent infinite query results, the length of the path must be greater than zero, as discussed in §4.3. To enforce this, the function $len(P)$ computes the minimum length of a query based on its path type. This function is defined recursively as follows:

$$len(\bot) := 0 \qquad len(P_1 + P_2) := \mathtt{min}(len(P_1), len(P_2)) \qquad len(N) := 0 \qquad len(N_1 - N_2) := 1$$

The resulting type environment is obtained by converting each variable type in Γ into a list type via the function $[\Gamma]^l$, defined as $[\cdot]^l := \cdot$ and $[\Gamma, x \mapsto T]^l := [\Gamma]^l, x \mapsto [T]^l$. To validate repetition, the path type is checked for self-concatenation using the refinement operator. If a pattern $(A) \longrightarrow (B)$ is repeated, it suffices to test whether $(A) \longrightarrow (B)(A) \longrightarrow (B)$ is concatenable. Additional concatenations do not provide further information and are therefore unnecessary. Thus, repetition is tested up to two times to avoid excessive conservatism, since n could be zero or one.

Expressions. The typing rules for expressions are presented at the bottom of Figure 7. Most rules are standard. The judgment $\Gamma \vdash t : \tau$ denotes that an expression t has type τ under the type environment Γ. Constants are typed using the function ty (rule (Tc)). For projection expressions $x.a$, typing is determined by retrieving the type of attribute a from the variable type $\Gamma(x)$ (rule (Tpj)). Attribute projection for variable types and property types is defined as follows:

$$\begin{aligned}
(\!|\ell\ R|\!)(a) &= (\_ \overset{\ell\ R}{\Longleftrightarrow} \_)(a) = R(a) & \{\overline{a_i : \tau_i}^i\}(a) &= \tau \quad \text{where } (a : \tau) \in \{\overline{a_i : \tau_i}^i\} \\
(T_1 + T_2)(a) &= T_1(a) \sqcup T_2(a) & \{\overline{a_i : \tau_i}^i\}(a) &= \bot \quad \text{otherwise} \\
[T]^l(a) &= [T(a)]^l & \{\overline{a_i : \tau_i}^i, \star\}(a) &= \tau \quad \text{where } (a : \tau) \in \{\overline{a_i : \tau_i}^i\} \\
\bot(a) &= \bot & \{\overline{a_i : \tau_i}^i, \star\}(a) &= \star \quad \text{otherwise}
\end{aligned}$$

If an attribute a is not found, the return type is $\bot$. Rules (Tbop) and (Tuop) handle binary and unary operations by using the meta-function Δ, which returns a type for a given operation. Each subterm is typed and checked against the expected domain type using the meet operator. The return type is determined using the $\widetilde{cod}$ meta-function, which returns the codomain of the function if none of the arguments is $\bot$; otherwise, it returns $\bot$, propagating the type inconsistency or error.

Example. Consider a schema with node types: $N_T = (\!|\mathtt{Person\&Teacher}\ \{\mathtt{status} : \mathtt{String}, \mathtt{name} : \mathtt{String}, \star\}|\!)$, $N_S = (\!|\mathtt{Person\&Student}\ \{\mathtt{name} : \star, \mathtt{status} : \mathbb{Z}\}|\!)$, and $N_? = (\!|\star\ \{\mathtt{status} : \mathbb{B}, \star\}|\!)$. The schema also has the edges types: $N_T \xrightarrow{\mathtt{Likes}\ \{\star\}} N_?$, $N_? \xrightarrow{\mathtt{Author}\ \{\star\}} N_S$, and $N_T \overset{\mathtt{Knows}\ \{\mathtt{since} : \mathbb{Z}\}}{\overline{\hspace{5em}}} N_S$.

The query pattern being type-checked consists of the following concatenation of a node pattern and a left-directed edge pattern $(x_1 : \texttt{Person}\ \{\texttt{name} : \star, \texttt{status} : \texttt{String}\})\ \xleftarrow{x_2:\texttt{Likes}\ \{\star\}}$.

We begin by type-checking the node pattern on the left,which by rule (Tnode) (Fig. 7), means applying the refinement operator. We observe that N_T is the only node type in the schema that is a subtype of $(\!|\texttt{Person}\ \{\texttt{name} : \star, \texttt{status} : \texttt{String}\}|\!)$. Indeed the type of $\texttt{status}$ is either $\mathbb{Z}$ or $\mathbb{B}$ for N_S and $N_?$, respectively, which are not subtypes of $\texttt{String}$. Since the label $\texttt{Person\&Teacher}$ and the record type are the subtype of $\texttt{Person}$ and the required fields, we select the most precise type resulting from the meet: $(\!|\texttt{Person\&Teacher}\ \{\texttt{name} : \texttt{String}, \texttt{status} : \texttt{String}\}|\!)$. Thus, the type environment is $x_1 \mapsto (\!|\texttt{Person\&Teacher}\ \{\texttt{name} : \texttt{String}, \texttt{status} : \texttt{String}\}|\!)$ and the path type is $(\!|\texttt{Person\&Teacher}\ \{\texttt{name} : \texttt{String}, \texttt{status} : \texttt{String}\}|\!)$, since $\lfloor\cdot\rfloor^{\Leftrightarrow}$ returns the node type itself.

Next, we analyze the edge pattern $\xleftarrow{x_2:\texttt{Likes}\ \{\star\}}$. This pattern can only be matched by the edge type $N_T \xrightarrow{\texttt{Likes}\ \{\star\}} N_?$ in the schema. Therefore, the resulting type environment is defined as $x_2 \mapsto N_T \xrightarrow{\texttt{Likes}\ \{\star\}} N_?$ and the path type becomes $N_? - N_T$.

However, a consistency check between the target node types of both path types reveals that $\texttt{String}$ and $\mathbb{B}$ are not compatible for the attribute $\texttt{status}$. As a result, the refined node type includes a $\bot$ for that attribute. The final path type is $(\!|\texttt{Person\&Teacher}\ \{\texttt{name} : \texttt{String}, \texttt{status} : \bot, \star\}|\!)$-$- N_T$, and the type environment is $x_1 \mapsto (\!|\texttt{Person\&Teacher}\ \{\texttt{name} : \texttt{String}, \texttt{status} : \texttt{String}, \star\}|\!)$, $x_2 \mapsto N_T \xrightarrow{\texttt{Likes}\ \{\star\}} N_?$.

6 Properties

This section presents key properties satisfied by FPPC. Establishing the metatheory requires introducing additional notation and relations. A fully precise family of data types $(\dot{\ell}, \dot{R}, ,\dot{L}, \dot{T}, \dot{N}, \dot{E})$ is introduced for inferring graph types. These types follow the same structure as their gradual counterparts but exclude unknown types, gradual unions, and open property types. A complete definition is provided in the Appendix. These fully precise types enable accurate type inference for nodes and edges, following the rules outlined below:

$$\frac{\vdash \mathcal{L}(n)\ \mathcal{P}(n) : \dot{L}}{G \vdash n : (\!|\dot{L}|\!)} \qquad \frac{\vdash \mathcal{L}(e)\ \mathcal{P}(e) : \dot{L} \qquad G \vdash \mathsf{src}(e) : \dot{N}_1 \qquad G \vdash \mathsf{tgt}(e) : \dot{N}_2}{G \vdash e : \dot{N}_1 \xrightarrow{\dot{L}} \dot{N}_2}$$

$$\frac{\vdash \mathcal{L}(e)\ \mathcal{P}(e) : \dot{L} \qquad \mathsf{endpoints}(e) = \{n_1, n_2\} \qquad G \vdash n_1 : \dot{N}_1 \qquad G \vdash n_2 : \dot{N}_2}{G \vdash e : \dot{N}_1 \overset{\dot{L}}{\sim} \dot{N}_2}$$

where $G = \langle \mathcal{N}_{id}, \mathcal{E}_d, \mathcal{E}_u, \mathcal{L}, \mathsf{endpoints}, \mathsf{src}, \mathsf{tgt}, \mathcal{P}\rangle$. With these rules established, a graph is considered well-formed with respect to a schema when each data element in the graph has a corresponding type in the schema.

Definition 6.1 (Well-formed Graph). $S \vdash G$ is defined iff:

- $\forall n \in N_{id}.\ G \vdash n : \dot{N}$ then $\exists N \in S.N \sim \dot{N}$.
- $\forall e \in E_d \cup E_u.\ G \vdash e : \dot{E}$ then $\exists E \in S.E \sim \dot{E}$.

The relation $A \sim B$ represent the gradual counterpart of equality and holds if $A \lesssim B$ and $B \lesssim A$ hold simultaneously. Using only subtyping is insufficient, as it violates the emptiness property (§6.2). To illustrate this, consider a graph with two nodes: one labeled A and another labeled A and B. If schemas that are supertypes of the data, such as $S = (\!|\top\ \{\star\}|\!)$, are allowed, then the type system would raise an empty-result warning for the pattern $(\texttt{A})$, as A is not a subtype of $\top$. However, executing this query would yield two paths instead.

6.1 Type Safety

The type safety property guarantees that well-typed path patterns do not get stuck during execution and that their evaluation always produces well-typed results (subtype of the original path pattern type). To type-check results, the notation γ is introduced to denote a result set $\{\overline{(p, \mu)}\}$. A result set γ is well-typed under graph G, if all paths p and assigments μ are well-typed under the graph.

Since paths and path types can have different lengths due to repetitions, their structures cannot be precisely matched. However, the first and last elements must align. The judgment $G \vdash p : (N_1, N_2)$ ensures that the first and last nodes of a path conform to the specified type and is defined as follows:

$$\frac{\vdash \mathcal{L}(n)\, \mathcal{P}(n) : \dot{L}}{G \vdash n : (\llparenthesis \dot{L} \rrparenthesis, \llparenthesis \dot{L} \rrparenthesis)} \qquad \frac{G \vdash n_1 : \dot{N}_1 \quad G \vdash n_2 : \dot{N}_2 \quad \mathsf{src}(e) = n_1 \quad \mathsf{tgt}(e) = n_2}{G \vdash n_1 e n_2 : (\dot{N}_1, \dot{N}_2)}$$

$$\frac{G \vdash n_1 : \dot{N}_1 \quad G \vdash n_2 : \dot{N}_2 \quad \mathsf{endpoints}(e) = \{n_1, n_2\}}{G \vdash n_1 e n_2 : (\dot{N}_1, \dot{N}_2)} \qquad \frac{G \vdash p_1 : (\dot{N}_{11}, \dot{N}) \quad G \vdash p_2 : (\dot{N}, \dot{N}_{22})}{G \vdash p_1 \cdot p_2 : (\dot{N}_{11}, \dot{N}_{22})}$$

To type assignments, the judgment $G \vdash r : T$ extends the previously defined typing rules for nodes and edges. This extension includes additional cases to account for lists and null values:

$$\frac{G \vdash r_i : \dot{T}}{G \vdash [r_i]^l : [\dot{T}]^l} \qquad \frac{}{G \vdash \mathsf{null} : \mathsf{Null}}$$

An assignment is well-typed to type environment Γ, under graph G, if for all variables in the domain, the values r in the codomain can be typed under graph G as a type variable T:

Definition 6.2 (Well-typed Assignments). $G \vdash \mu : \Gamma$ iff $\forall x \in dom(\mu), G \vdash \mu(x) : T$ and $T \lesssim \Gamma(x)$.

Finally, type safety is defined as follows:

Theorem 6.3 (Soundness). *If* $S \vdash \pi : P;\ \Gamma$ *and* $S \vdash G$ *then* $\exists \gamma. [\![\pi]\!]_G = \gamma, \forall (p, \mu) \in \gamma, G \vdash \mu : \Gamma$, $G \vdash p : (N_1, N_2)$ *and* $(N_1, N_2) \lesssim (\mathsf{first}(P), \mathsf{last}(P))$.

6.2 Emptiness

The type system of FPPC under-approximates empty results. To predict whether a path pattern will evaluate to an empty result, the empty$(\cdot)$ meta-function is introduced. This function applies to path types and variable types and is defined inductively (the definition can be found in Appendix). Intuitively, this metafunction checks whether $\perp$ is present in some subcomponent. If $\perp$ is found, the function returns true. The only exception is the case of union types. Here, the function returns true only if both subcomponents contain $\perp$. If only one subcomponent contains a type error, the entire union may still yield a non-empty result. A type environment is considered empty if at least one variable type in its codomain is empty.

Definition 6.4 (Empty Type Environments). $\mathsf{empty}(\Gamma)$ if $\exists x \in dom(\Gamma), \mathsf{empty}(\Gamma(x))$

Finally, emptiness is defined such that, given a well-typed path pattern, if either the path type is empty or the type environment is empty, then the result is definitively empty.

Theorem 6.5 (Emptiness). *If* $S \vdash \pi : P;\ \Gamma, S \vdash G$ *and (*$\mathsf{empty}(P)$ *or* $\mathsf{empty}(\Gamma)$*) then* $[\![\pi]\!]_G = \{\}$.

The proof relies on the auxiliary Lemma 6.6, which ensures that if two node types, N_2 and N_4, are inconsistent under a schema S, then their corresponding subtyping data node types, $\dot{N}_1$ and $\dot{N}_3$, must also be inconsistent under the same schema S.

Lemma 6.6. *If* $\dot{N}_1 \lesssim N_2, \dot{N}_3 \lesssim N_4, S \vdash N_2 \sqcap N_4 \triangleright N$ *and* $\mathsf{empty}(N)$ *then* $S \vdash \dot{N}_1 \sqcap \dot{N}_3 \triangleright N'$ *and* $\mathsf{empty}(N')$.

The converse of Theorem 6.5 clearly does not hold, as it is possible for patterns to yield empty results purely by coincidence, without any underlying type inconsistency. To further illustrate this, consider a variant of the semantics where type errors are explicitly propagated. In this setting, Theorem 6.5 fails to hold, since evaluation may result in a type error rather than an empty result. One might conjecture that in this setting, a well-typed pattern reducing to a type error implies that either the path type or the type environment of the pattern is empty. However, this intuition does not hold. For example, consider a union pattern of the form $\pi_1 + \pi_2$. If π_1 reduces to an error, then the entire union $\pi_1 + \pi_2$ also reduces to an error, even if π_2 evaluates successfully. In such cases, the resulting path type or type environment of the overall pattern may still be non-empty, contradicting the statement above. However, if we instead adopt an alternative semantics—where a union pattern $\pi_1 + \pi_2$ evaluates to the result of π_2 when π_1 fails with an error, then the conjecture might hold. This suggests that the treatment of type errors in the evaluation semantics plays a critical role in determining whether such implications are valid.

6.3 Gradual Guarantee

The gradual guarantee is a fundamental property of gradual typing, first introduced by Siek et al. [2015]. It comprises two key principles: the static gradual guarantee (SGG) and the dynamic gradual guarantee (DGG). The SGG ensures that typing remains monotonic with respect to precision, while the DGG guarantees that reduction preserves monotonicity with respect to precision. To formally define the gradual guarantee for FPPC, we begin by establishing precision relations for types. Specifically, a property type R is considered more precise than R' if their corresponding attributes exhibit precision-related types:

$$\frac{\tau_i \sqsubseteq \tau_i'}{\{\overline{a_i : \tau_i}^i\} \sqsubseteq \{\overline{a_i : \tau_i'}^i\}} \qquad \frac{\{\overline{a_i : \tau_i}^i\} \sqsubseteq \{\overline{a_j : \tau_i'}^i\}}{\{\overline{a_i : \tau_i}^i, \overline{a_k : \tau_k}^k, *\} \sqsubseteq \{\overline{a_i : \tau_i'}^i, \star\}}$$

where $\{\overline{a_i : \tau_i}^i, *\}$ simultaneously denotes close $\{\overline{a_i : \tau_i}^i\}$ and open records $\{\overline{a_i : \tau_i}^i, \star\}$. The precision of simple types $\tau \sqsubseteq \tau'$ is standard. In this context, the bottom type $\bot$ represents an error state, making it more precise than any other type. Formally, the definition is given as follows:

$$\frac{}{\iota \sqsubseteq \iota} \qquad \frac{}{\tau \sqsubseteq \star} \qquad \frac{\tau_1 \sqsubseteq \tau \quad \tau_2 \sqsubseteq \tau}{\tau_1 + \tau_2 \sqsubseteq \tau} \qquad \frac{\tau \sqsubseteq \tau_1}{\tau \sqsubseteq \tau_1 + \tau_2} \qquad \frac{\tau \sqsubseteq \tau_2}{\tau \sqsubseteq \tau_1 + \tau_2} \qquad \frac{}{\bot \sqsubseteq \tau}$$

The precision relation for labels, denoted as $\ell_i \sqsubseteq \ell_j$, is defined analogously to that of simple types:

$$\frac{}{1 \sqsubseteq 1} \qquad \frac{}{\ell \sqsubseteq \star} \qquad \frac{\ell_1 \sqsubseteq \ell_3 \quad \ell_2 \sqsubseteq \ell_4}{\ell_1 \& \ell_2 \sqsubseteq \ell_3 \& \ell_4} \qquad \frac{\ell_1 \sqsubseteq \ell_2 \quad \ell_1 \sqsubseteq \ell_3}{\ell_1 \sqsubseteq \ell_2 + \ell_3} \qquad \frac{\ell_1 \sqsubseteq \ell_3}{\ell_1 + \ell_2 \sqsubseteq \ell_3}$$

$$\frac{\ell_2 \sqsubseteq \ell_3}{\ell_1 + \ell_2 \sqsubseteq \ell_3} \qquad \frac{}{\varepsilon \sqsubseteq \varepsilon} \qquad \frac{}{\bot \sqsubseteq \ell}$$

The precision relations for variable types, terms, path patterns, and path types are straightforward and defined inductively (see Appendix for their formal definitions). Additionally, a schema S_1 is more precise than S_2 if their corresponding node and edge types adhere to the precision relation. Using these precision relations, we establish the static gradual guarantee (SGG) for FPPC in Theorem 6.7.

Theorem 6.7 (Static Gradual Guarantee). *If $S_1 \sqsubseteq S_2$, $\pi_1 \sqsubseteq \pi_2$ and $S_1 \vdash \pi_1 : P_1;\ \Gamma_1$ then $S_2 \vdash \pi_2 : P_2;\ \Gamma_2$, $P_1 \sqsubseteq P_2$ and $\Gamma_1 \sqsubseteq \Gamma_2$.*

The dynamic gradual guarantee (DGG) for FPPC is established in Theorem 6.8. This theorem states that, given two well-formed schemas related by precision and two well-typed patterns also related by precision, the result set produced by evaluating the more precise pattern must be contained within the result set produced by evaluating the less precise one. However, type tests t is τ and casts t as τ can violate the DGG. For example, if `(true as ℕ)` reduces to 1 and `(false as ℕ)` to 0, then the term `t = (('A' is 𝔹) as ★) as ℕ ≤ 0` reduces to `true`, while the less precise term `t' = (('A' is ★) as ★) as ℕ ≤ 0` reduces to `false`. As a result, the outputs of $\pi_{\langle t\rangle}$ are not necessarily contained within those of $\pi'_{\langle t'\rangle}$. To address this issue, we restrict the DGG to patterns related by a stricter precision relation, denoted $\sqsubseteq^*$. Specifically, we require that type tests and casts be invariant in their underlying type argument, while otherwise following the standard $\sqsubseteq$ relation. The following excerpted rules illustrate the precision constraints:

$$\frac{\pi \sqsubseteq^* \pi' \qquad t \sqsubseteq^* t'}{\pi_{\langle t\rangle} \sqsubseteq^* \pi'_{\langle t'\rangle}} \qquad \frac{d \sqsubseteq d'}{(d) \sqsubseteq^* (d')} \qquad \frac{t \sqsubseteq^* t'}{t \text{ as } \tau \sqsubseteq^* t' \text{ as } \tau} \qquad \frac{t \sqsubseteq^* t'}{t \text{ is } \tau \sqsubseteq^* t' \text{ is } \tau}$$

Theorem 6.8 (Dynamic Gradual Guarantee). *If* $S_1 \sqsubseteq S_2$, $S_1 \vdash G$, $S_2 \vdash G$, $S_1 \vdash \pi_1 : P_1;\ \Gamma_1$, $S_2 \vdash \pi_2 : P_2;\ \Gamma_2$, $\pi_1 \sqsubseteq^* \pi_2$ *then* $[\![\pi_1]\!]_G \sqsubseteq [\![\pi_2]\!]_G$.

7 Prototype Implementation

To validate the formal development of FPPC and support interactive experimentation with typed graph queries, we implemented a Python prototype. Written in Python v3.10+, the prototype consists of around 3,000 lines of code and closely follows the structure of the calculus presented in this paper. It provides end-to-end support for parsing, typechecking, and executing path patterns over concrete graph databases.

Core Components. The implementation consists of three main components: a parser for a fragment of the GQL path pattern language extended with gradual types and property-based filtering; a static typechecker implementing the rules of FPPC; and a runtime engine that evaluates queries against a graph database.

Syntax and Language Fragment. The prototype adopts a concrete syntax close to GQL, with adjustments for features of FPPC. In particular: union patterns use `P | Q` instead of `P + Q`; filters can be used inside descriptors: `(x WHERE cond)` compiles to `(x) WHERE cond`; property types distinguish between open and closed records using notation `{...}` and `{{...}}` repectively; edges are written using the form `-[d]->`, `<-[d]-`, `-[d]-`, and `~[d]~`.

Typing and Error Reporting. Typechecking is performed with respect to a schema, which is fully imprecise by default. It can be constructed manually or inferred automatically by the prototype from a JSON-encoded graph database. The typechecker is designed to be non-failing: rather than terminating at the first type error, it continues checking optimistically, accumulating both errors and warnings. This design supports informative feedback and allows the analysis of partially valid queries. For instance, consider the pattern

```
(x: {status: bool} WHERE x.status > 0) -[:Author WHERE y.foo]->
```

This query is syntactically valid, but contains both a type inconsistency and a reference to an unbound variable. The typechecker emits the following diagnostics:

```
Path: ⊥
Context: x ↦ ⦇★ {status:bool,★}⦈
Warnings:
Binop > between types bool and int is not defined
..at check_expr: ⊢ (x.status > 0) : ???
Filter expression (x.status > 0) has type ⊥, which is definitively not a bool.
.at check_path_pattern: ⊢ ⦇x: ★ {status:bool,★}⦈ WHERE (x.status > 0) : ???
Errors:
Variable y not found in context
..at check_expr: ⊢ y.foo : ???
```

Evaluation. Once a query is validated, it can be evaluated over a graph database. The runtime follows the reduction semantics of FPPC and produces all matching paths and variable bindings. The semantics faithfully implement the formal behavior described in Figures 4 and 5.

Limitations. The runtime disallows unbounded repetition in path patterns to avoid non-termination; all repetitions require explicit bounds.

Overall, the prototype demonstrates that FPPC can be realized in practice. While the current prototype focuses on path patterns—a core fragment of the GQL query model—it lays the foundation for broader adoption. Future work is needed to support the full range of GQL queries.

8 Related Work

GQL. The Graph Query Language (GQL) [ISO/IEC 2024] offers a unified framework for querying graph databases, analogous to SQL for relational models. GQL builds on prior work, particularly in property graph schemas and graph pattern matching. Francis et al. [2023b] and Deutsch et al. [2022] provide a comprehensive digest of GQL, outlining its formal syntax, multi-graph querying capabilities, and integration with existing graph database systems. Francis et al. [2023a] proposes the Graph Pattern Calculus (GPC), a theoretical framework that formalizes pattern syntax, semantics, and typing rules, thus capturing the key pattern-matching features of the emerging GQL and SQL/PGQ standards. FPPC extends the syntax of GPC by adding property type filtering, and refines the typing rules, as shown in Section 5. Complementary to these efforts, PG-Schema [Angles et al. 2023] introduces a schema language for property graphs, enhancing GQL's schema validation through structural constraints and adding a rich typing system for defining valid graphs. However, it does not consider how node and edge specifications can be incorporated to aid in GQL query evaluation, as we do in this paper. Our work can be viewed as an extension of PG-Schema to the query level, shifting from solely validating stored data to schema-aware query validation through typing, while also enhancing flexibility via the introduction of gradual types. A rich body of theoretical work on the complexity of querying regular graph patterns has also contributed to fundamental design decisions in GQL, emphasizing automata-based techniques for efficient query processing [Barceló et al. 2014; Barceló Baeza 2013; Libkin et al. 2016; Reutter et al. 2017].

Gradual Typing. Gradual typing, first introduced by Siek and Taha [2006], provides a framework that seamlessly integrates static and dynamic typing, allowing programmers to selectively apply type annotations while maintaining type safety. A more formal foundation for gradual typing was established via the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016], which systematically derives gradual type systems from their static counterparts using abstract interpretation techniques. Similar to Ye et al. [2023], FPPC also leverages AGT to justify relations, and draws inspiration from Bañados Schwerter et al. [2021] and Ye et al. [2024] to handle extensible records and intersections, benefiting from gradual typing through the use of the unknown type $\star$ to support an incremental design.

Castagna and Lanvin [2017] and Castagna et al. [2019] explored intersection and union types with gradual typing using a set-theoretic approach. In the former, negation is restricted to static types, while in the latter, it relies on solver-based subtyping. We decided to follow the AGT approach because the GQL ISO standard defines both open and closed records, and their gradual counterparts are well studied in the AGT literature [Bañados Schwerter et al. 2021; Garcia et al. 2016]. Thus our type system combines multiple features—not only set-theoretic types, but also gradual records. We believe that the combination of different features is naturally handled under AGT. Another reason is that to adapt Castagna et al.'s set-theoretic approach, we believe we would likely need to redefine the semantics entirely to match a different theory. That would move us too far away from the GQL ISO standard, which is one of the guidelines of FPPC. Furthermore, this would introduce additional slow down during typechecking due to constraint solving, which could compete with query execution itself. CDuce is also a strong system for semi-structured data, but we did not adopt it because it does not align well with how GQL and GPC works. CDuce uses tagged unions and requires pattern-matching to distinguish cases. Moreover, its notion of consistency is stricter than FPPC. For instance, in CDuce, $A + B$ is not considered a consistent subtype of A. In contrast, in FPPC, it is because if something is optimistically of type $A + B$, it may still be an A. This flexibility is necessary to under-approximate empty results for queries such as `(x: a: str + 𝔹 WHERE x.a > 0)`.

Under-approximating Errors. In FPPC, well-typed queries that generate empty-result warnings indeed return empty results due to type inconsistencies (under-approximation). However, well-typed queries never produce runtime errors or become stuck (over-approximation). Several related works have explored under-approximation as a principled strategy for detecting errors. Incorrectness Logic (IL) [O'Hearn 2019] is a novel formalism that provides a dual perspective to Hoare logic by focusing on proving the presence of bugs rather than their absence. This approach has been particularly influential in the development of automated bug detection tools, leveraging under-approximation techniques to identify provable error states. Building on this foundation, Le et al. [2022] proposes Pulse-X, an automatic program analysis tool based on Incorrectness Separation Logic (ISL), a synthesis of IL and separation logic, which detected 15 previously unknown bugs in OpenSSL. Additionally, Zhou et al. [2023] explores the application of under-approximate reasoning principles in the context of property-based testing, proposing a refinement-type system that guarantees exhaustive coverage of test input generators. Similarly, Lindahl and Sagonas [2006] introduces *success typings*, an under-approximate approach to static analysis that detects definite type errors in Erlang. FPPC aligns with these efforts by statically preventing empty query results via the empty predicate, representing a form of under-approximate error detection. A related goal is pursued by Seidel et al. [2016], who address static type errors through a dynamic approach. Their system uses symbolic execution on ill-typed programs to synthesize counterexamples that cause the program to go wrong. This method is under-approximate: it may fail to produce a witness, but any witness it does produce corresponds to a real error. Finally, the type system of FPPC shares a conceptual goal with the work of Hriţcu [2011]: detecting type-based inconsistencies via disjointness reasoning. However, while Hriţcu rely on semantic proofs or logical entailments to establish disjointness, FPPC achieves this purely syntactically within its type environment.

9 Conclusion

Traditional query languages often overlook type systems. This work shows that a flexible type system can enhance query reliability by detecting type inconsistencies early. Applying typing tools to GQL can improve robustness and usability, making query languages more reliable and accessible.

We introduce FPPC, an extension of GQL's pattern calculus (GPC) with property-based filtering and a flexible type system using schemas. Our approach proves important theoretical properties, such as emptiness (detecting queries that always return empty results) and type safety (ensuring well-typed queries do not fail at runtime). Additionally, we establish gradual guarantees, enabling developers to smoothly transition between untyped and typed queries and schemas. This work targets a core fragment of GQL, omitting features such as aggregation and negation in label predicates. Negation is particularly challenging due to its interaction with subtyping and the unknown type, a difficulty that future work may address using the set-theoretic approach of Castagna and Lanvin [2017]. Our research also opens the door to many potential developments not only in programming languages. From a database point of view, the challenges include to explore optimizations for query evaluation under these typing enhancements, and further refinements to the interaction between schema evolution and gradual typing in graph databases.

Data-Availability Statement

We developed a Python prototype for FPPC. It includes a type checker and interpreter for path queries, and supports optional schema inference. The tool checks queries, emits warnings for type inconsistencies, detects statically empty queries, and can execute valid ones against JSON-based databases. The artifact that supports the paper is available on Zenodo [Ye et al. 2025].

References

Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. 2023. PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* 1, 2, Article 198 (June 2023), 25 pages. doi:10.1145/3589778

Renzo Angles and Claudio Gutiérrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (2008), 1:1–1:39. doi:10.1145/1322432.1322433

Oana Balalau, Pablo Bertaud-Velten, Younes El Fraihi, Garima Gaur, Oana Goga, Samuel Guimaraes, Ioana Manolescu, and Brahim Saadi. 2024. FactCheckBureau: Build Your Own Fact-Check Analysis Pipeline. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management, CIKM 2024, Boise, ID, USA, October 21-25, 2024*, Edoardo Serra and Francesca Spezzano (Eds.). ACM, 5185–5189. doi:10.1145/3627673.3679220

Felipe Bañados Schwerter, Alison M Clark, Khurram A Jafery, and Ronald Garcia. 2021. Abstracting gradual typing moving forward: Precise and space-efficient. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

Pablo Barceló, Leonid Libkin, and Juan L. Reutter. 2014. Querying Regular Graph Patterns. *J. ACM* 61, 1, Article 8 (Jan. 2014), 54 pages. doi:10.1145/2559905

Pablo Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (New York, New York, USA) *(PODS '13)*. Association for Computing Machinery, New York, NY, USA, 175–188. doi:10.1145/2463664.2465216

Stefan Brass and Christian Goldberg. 2006. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software* 79, 5 (2006), 630–644. doi:10.1016/j.jss.2005.06.028 Quality Software.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. doi:10.1145/3110285

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual typing: a new perspective. *Proc. ACM Program. Lang.* 3, POPL, Article 16 (Jan. 2019), 32 pages. doi:10.1145/3290329

Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. https://www.w3.org/TR/rdf11-bconcepts/

Alin Deutsch, Nadime Francis, Alastair Green, Keith W. 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 Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2246–2258. doi:10.1145/3514221.3526057

Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023a. GPC: A Pattern Calculus for Property Graphs. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Seattle, WA, USA) *(PODS '23)*. Association for Computing Machinery, New York, NY, USA, 241–250. doi:10.1145/3584372.3588662

Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023b. A Researcher's Digest of GQL. In *The 26th International Conference on Database Theory, 2023.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 1–1.

Ronald Garcia, Alison M Clark, and Éric Tanter. 2016. Abstracting gradual typing. *ACM SIGPLAN Notices* 51, 1 (2016), 429–442. doi:10.1145/2837614.2837670

Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-bquery/

Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. *Knowledge Graphs.* Morgan & Claypool Publishers. doi:10.2200/S01125ED1V01Y202109DSK022

Cătălin Hriţcu. 2011. *Union, Intersection and Refinement Types and Reasoning About Type Disjointness for Security Protocol Analysis.* Ph. D. Dissertation. Saarland University, Saarbrücken, Germany.

ISO/IEC. 2024. ISO/IEC 39075:2024 Information technology — Database languages — GQL. Available online: https://www.iso.org/standard/76120.html. Accessed: 2024-05-15.

Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 46, 12 pages. doi:10.1145/3597503.3623307

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (April 2022), 27 pages. doi:10.1145/3527325

Leonid Libkin, Wim Martens, and Domagoj Vrgoc. 2016. Querying Graphs with Data. *J. ACM* 63, 2, Article 14 (March 2016), 53 pages. doi:10.1145/2850413

Tobias Lindahl and Konstantinos Sagonas. 2006. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 167–178.

Memgraph Team. 2023. Memgraph. https://memgraph.com/

Peter W O'Hearn. 2019. Incorrectness logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32. doi:10.1145/3371078

Nicole Redaschi and UniProt Consortium. 2009. UniProt in RDF: tackling data integration and distributed annotation with the semantic web. *Nature precedings* (2009), 1–1.

Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2017. Regular Queries on Graph Databases. *Theory Comput. Syst.* 61, 1 (2017), 31–83. doi:10.1007/S00224-b016-b9676-b2

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). *SIGPLAN Not.* 51, 9 (Sept. 2016), 228–242. doi:10.1145/3022670.2951915

Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming*.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 274–293. doi:10.4230/LIPIcs.SNAPL.2015.274

TigerGraph Team. 2021. TigerGraph Documentation – version 3.1. https://docs.tigergraph.com/

Matías Toro and Éric Tanter. 2017. A gradual interpretation of union types. In *Static Analysis: 24th International Symposium, SAS 2017, New York, NY, USA, August 30–September 1, 2017, Proceedings 24*. Springer, 382–404. doi:10.1007/978-b3-b319-b66706-b5_19

Vesoft Inc/Nebula. 2023. NebulaGraph. https://www.nebula-bgraph.io/

Jim Webber. 2012. A programmatic introduction to Neo4j. In *SPLASH*. doi:10.1145/2384716.2384777

Wenjia Ye, Bruno C. d. S. Oliveira, and Matías Toro. 2024. Merging Gradual Typing. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 294 (Oct. 2024), 29 pages. doi:10.1145/3689734

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 (Artifact)*. doi:10.5281/zenodo.16909264

Wenjia Ye, Matías Toro, and Federico Olmedo. 2023. A Gradual Probabilistic Lambda Calculus. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (Oct. 2023), 256–285. doi:10.1145/3586036

Zhe Zhou, Ashish Mishra, Benjamin Delaware, and Suresh Jagannathan. 2023. Covering all the bases: Type-based verification of test input generators. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1244–1267. https://doi.org/10.1145/3591271

Received 2025-03-24; accepted 2025-08-12