Common Foundations for SHACL, ShEx, and PG-Schema

Shqiponja Ahmetaj
shqiponja.ahmetaj@tuwien.ac.at
TU Wien
Vienna, Austria

Iovka Boneva
iovka.boneva@univ-lille.fr
Univ. Lille, CNRS, Inria, Centrale Lille,
UMR 9189 CRIStAL
Lille, France

Jan Hidders
j.hidders@bbk.ac.uk
Birkbeck, University of London
London, UK

Katja Hose
katja.hose@tuwien.ac.at
TU Wien
Vienna, Austria

Maxime Jakubowski
maxime.jakubowski@tuwien.ac.at
TU Wien
Vienna, Austria

Jose Emilio Labra Gayo
labra@uniovi.es
University of Oviedo
Oviedo, Spain

Wim Martens
wim.martens@uni-bayreuth.de
University of Bayreuth
Bayreuth, Germany

Fabio Mogavero
fabio.mogavero@unina.it
Università di Napoli Federico II
Naples, Italy

Filip Murlak
f.murlak@uw.edu.pl
University of Warsaw
Warsaw, Poland

Cem Okulmus
cem.okulmus@uni-paderborn.de
Paderborn University
Paderborn, Germany

Axel Polleres
axel.polleres@wu.ac.at
WU Wien
Vienna, Austria
CSH Vienna
Vienna, Austria

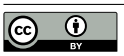
Ognjen Savković
ognjen.savkovic@unibz.it
Free University of Bolzano
Bolzano, Italy

Mantas Šimkus
mantas.simkus@tuwien.ac.at
TU Wien
Vienna, Austria

Dominik Tomaszuk
d.tomaszuk@uwb.edu.pl
TU Wien
Vienna, Austria
University of Bialystok
Bialystok, Poland

Abstract

Graphs have emerged as a foundation for a variety of applications, including capturing factual knowledge, semantic data integration, social networks, and informing machine learning algorithms. Formalising properties of the data and ensuring data quality requires describing *schemas* of such graphs. Driven by diverse applications, the Semantic Web and database communities developed not only different graph data models—RDF and property graphs—but also different *graph schema languages*—SHACL, ShEx, and PG-Schema. Each language has its unique approach to defining constraints and validating graph data, leaving potential users in the dark about their commonalities and differences. In this paper, we provide concise formal definitions of the *core components* of these languages, employ a uniform framework to facilitate a comprehensive comparison between them, and identify a common set of functionalities, shedding light on both overlapping and distinctive features.



WWW '25, Sydney, NSW, Australia

ACM ISBN 979-8-4007-1274-6/25/04
https://doi.org/10.1145/3696410.3714694

CCS Concepts

• **Information systems → Graph-based database models**; **Semantic web description languages**.

Keywords

Graph Schema Languages, SHACL, ShEx, PG-Schema, RDF, Property Graphs, Graph Databases, Common Data Model

ACM Reference Format:
Shqiponja Ahmetaj, Iovka Boneva, Jan Hidders, Katja Hose, Maxime Jakubowski, Jose Emilio Labra Gayo, Wim Martens, Fabio Mogavero, Filip Murlak, Cem Okulmus, Axel Polleres, Ognjen Savković, Mantas Šimkus, and Dominik Tomaszuk. 2025. Common Foundations for SHACL, ShEx, and PG-Schema. In *Proceedings of the ACM Web Conference 2025 (WWW '25), April 28-May 2, 2025, Sydney, NSW, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3696410.3714694

1 Introduction

Driven by the unprecedented growth of interconnected data, *graph-based data representations* have emerged as an expressive and versatile framework for modelling and analysing connections in data sets [46]. This rapid growth however, has led to a proliferation of diverse approaches, each with its own identity and perspective.

The two most prominent graph data models are *RDF* (Resource Description Framework) [15] and *Property Graphs* [10]. In RDF, data is modelled as a collection of triples, each consisting of a subject, predicate, and object. Such triples naturally represent either edges in a directed labelled graph (where the predicates represent relationships between nodes), or attributes-value pairs of nodes. That is, objects can both be entities or atomic (literal) values. In contrast, Property Graphs model data as nodes and edges, where both can have labels and records attached, allowing for a flexible representation of attributes directly on the entities and relationships.

Similarly to the different data models, we are also seeing different approaches towards *schema languages* for graph-structured data. Traditionally, in the Semantic Web community, schema and constraint languages have been *descriptive*, focusing on flexibility to accommodate varying structures. However, there has been a growing need for more *prescriptive* schemas that focus on *validation of data*. At the same time, in the Database community, schemas have traditionally been prescriptive but, since the rise of semi-structured data, the demand for descriptive schemas has been growing. Thus, the philosophies of schemas in the two communities have been growing closer together.

For RDF, there are two main schema languages: SHACL (Shapes Constraint Language) [28], which is also a W3C recommendation, and ShEx (Shape Expressions) [43]. In the realm of Property Graphs, the current main approach is PG-Schema [3, 4]; it was developed with liaisons to the GQL and SQL/PGQ standardization committees and is currently being used as a basis for extending these standards. The development processes of these languages have been quite different. For SHACL and ShEx, the formal semantics were only introduced after their initial implementations, echoing the evolution of programming languages. Indeed, an analysis of SHACL's expressive power and associated decision problems appeared in the literature [7, 8, 34, 39–41] only after it was published as a W3C recommendation, leading up to a fully *recursive variant* of the language [2, 6, 13, 14, 40], whose semantics had been left undefined in the standard. A similar scenario occurred with ShEx, where formal analyses were only conducted in later phases [9, 49]. PG-Schema developed in the opposite direction. Here, a group of experts from industry and academia first defined the main ideas in a sequence of research papers [3, 4] and the implementation is expected to follow.

Since these three languages have been developed in different communities, in the course of different processes, it is no surprise that they are quite different. SHACL, ShEx, and PG-Schema use an array of diverse approaches for defining how their components work, ranging from *declarative* (formulae that *specify what to look for*) to *generative* (expressions that *generate the matching content*), and even combinations thereof. The bottom line is that we are left with three approaches to express a "schema for graph-structured data" that are very different at first glance.

As a group of authors coming from both the Semantic Web and Database communities, we believe that there is a *need for common understanding*. While the functionalities of schemas and constraints used in the two communities largely overlap, it is a daunting task to understand the essence of languages, such as SHACL, ShEx, and PG-Schema. In this paper, we therefore aim to shed light on the common aspects and the differences between these three languages. We focus on non-recursive schemas, as neither PG-Schema nor standard SHACL support recursion and also in the academic community the discussion on the semantics of recursive SHACL has not reached consensus yet [2, 6, 13, 14, 38, 40].

Using a common framework, we provide crisp definitions of the core components of the languages. Since the languages operate on different data models, as a first step we introduce the *Common Graph Data Model*, a mathematical representation of data that *canonically embeds* into both RDF and Property Graphs, and develop general common foundations (see Section 2). Precise abstractions of the three languages are presented in Sections 3 (SHACL), 4 (ShEx), and 5 (PG-Schema); in the full version of the paper we explain how and why we sometimes deviate from the original formalisms [1]. Each of these sections contains examples to give readers an immediate intuition about what kinds of conditions each language can express. Then, in Section 6, we present the *Common Graph Schema Language (CoGSL)*, which comprises features shared by them all.

Casting all three languages in a common framework has the immediate advantage that the reader can identify common functionalities *based on the syntax only*: on the one hand, we aim at giving the same semantics to schema language components that syntactically look the same, and on the other hand, we can provide examples of properties that distinguish the three languages using simple syntactic constructs that are not part of the common core. Aside from corner cases, properties expressed using constructs outside the common core are generally not expressible in all three languages. By providing an understanding of fundamental differences and similarities between the three schema languages, we hope to benefit both practitioners in choosing a schema language fitting their needs, and researchers in studying the complexity and expressiveness of schema languages.

2 Foundations

In this section we present some material that we will need in the subsequent sections, and define a data model that consists of common aspects of RDF and Property Graphs.

2.1 A Common Data Model

When developing a common framework for SHACL, ShEx, and PG-Schema, the first challenge is establishing a *common data model*, since SHACL and ShEx work on RDF, whereas PG-Schema works on Property Graphs. Rather than using a model that generalises both RDF and Property Graphs, we propose a simple model, called *common graphs*, which we obtained by asking what, fundamentally, are the *common aspects* of RDF and Property Graphs (Appendix B gives more details on the distilling of common graphs).

Let us assume disjoint countable sets of nodes $\mathcal{N}$, values $\mathcal{V}$, predicates $\mathcal{P}$, and keys $\mathcal{K}$ (sometimes called properties).

Definition 1. *A* common graph *is a pair* $\mathcal{G} = (E, \rho)$ *where*

- $E \subseteq_{fin} \mathcal{N} \times \mathcal{P} \times \mathcal{N}$ *is its set of edges (which carry predicates), and*
- $\rho\colon \mathcal{N} \times \mathcal{K} \rightharpoonup \mathcal{V}$ *is a finite-domain partial function mapping node-key pairs to values.*

The set of nodes of a common graph $\mathcal{G}$*, written* $\mathsf{Nodes}(\mathcal{G})$*, consists of all elements of* $\mathcal{N}$ *that occur in* E *or in the domain of* ρ*. Similarly,* $\mathsf{Keys}(\mathcal{G})$ *is the subset of* $\mathcal{K}$ *that is used in* ρ*, and* $\mathsf{Values}(\mathcal{G})$ *is the subset of* $\mathcal{V}$ *that is used in* ρ *(that is, the range of* ρ*).*

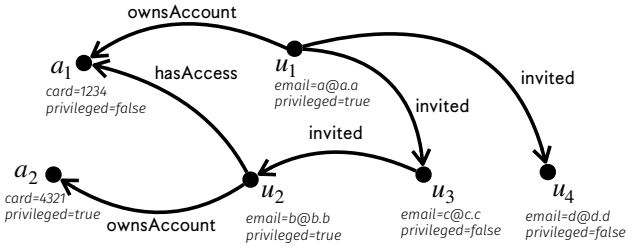


Figure 1: The media service common graph.

Example 1. Consider Figure 1, containing a graph to store information about *users* who may have access to (possibly multiple) *accounts* in, *e.g.*, a media streaming service. In this example, we have six nodes describing four persons (u_1, ..., u_4) and two accounts (a_1, a_2). As a common graph $\mathcal{G} = (E, \rho)$, the nodes are a_1, u_1, etc. Examples of edges in E are $(u_2, \text{hasAcccess}, a_1)$ and $(u_3, \text{invited}, u_2)$. Furthermore, we have $\rho(u_2, \mathit{email}) = \text{d@d.d}$ and $\rho(a_1, \mathit{card}) = 1234$. So, E captures the arrows in the figure (labelled with predicates) and ρ captures the key/value information for each node. Notice that a person may be the owner of an account, and may potentially have access to other accounts. This is captured using the predicates ownsAccount and hasAcccess, respectively. In addition, the system implements an invitation functionality, where users may invite other people to join the platform. The previous invitations are recorded using the predicate invited. Both accounts and users may be privileged, which is stored via a Boolean value of the key *privileged*. We note that the presence of the key *email* (*resp.*, of the key (credit) *card*) is associated with, and indeed identifies users (*resp.*, accounts).

It is easy to see that every common graph is a property graph (as per the formal definition of property graphs [3]). A common graph can also be seen as a set of triples, as in RDF. Let

$$\mathcal{E} = (\mathcal{N} \times \mathcal{P} \times \mathcal{N}) \ \cup \ (\mathcal{N} \times \mathcal{K} \times \mathcal{V}) \ .$$

Then, a common graph can be seen as a finite set $\mathcal{G} \subseteq \mathcal{E}$ such that for each $u \in \mathcal{N}$ and $k \in \mathcal{K}$ there is at most one $v \in \mathcal{V}$ such that $(u, k, v) \in \mathcal{G}$. Indeed, a common graph (E, ρ) corresponds to

$$E \ \cup \ \{(u, k, v) \mid \rho(u, k) = v\} \ .$$

When we write $\rho(u, k) = v$ we assume that ρ is defined on (u, k).

Throughout the paper we see property graph $\mathcal{G}$ simultaneously as a pair (E, ρ) and as a set of triples from $\mathcal{E}$, switching between these perspectives depending on what is most convenient at a given moment.

2.2 Node Contents and Neighbourhoods

Let $\mathcal{R}$ be the set of all *records*, *i.e.*, finite-domain partial functions $r : \mathcal{K} \rightharpoonup \mathcal{V}$. We write records as sets of pairs $\{(k_1, w_1), \ldots (k_n, w_n)\}$ where $k_1, \ldots, k_n$ are all different, meaning that k_i is mapped to w_i.

For a common graph $\mathcal{G} = (E, \rho)$ and node v in $\mathcal{G}$, by a slight abuse of notation we write $\rho(v)$ for the record $\{(k, w) \mid \rho(v, k) = w\}$ that collects all key-value pairs associated with node v in $\mathcal{G}$. We call $\rho(v)$ the *content* of node v in $\mathcal{G}$. This is how PG-Schema interprets common graphs: it views key-value pairs in $\rho(v)$ as *properties* of the node v, rather than independent, navigable objects in the graph.

SHACL and ShEx, on the other hand, view common graphs as sets of triples and make little distinction between keys and predicates. The following notion—when applied to a node—uniformly captures the local context of this node from that perspective: the content of the node and all edges incident with the node.

Definition 2 (Neighbourhood). *Given a common graph $\mathcal{G}$ and a node or value $v \in \mathcal{N} \cup \mathcal{V}$, the* neighbourhood *of v in $\mathcal{G}$ is* $\mathsf{Neigh}_{\mathcal{G}}(v) = \{(u_1, p, u_2) \in \mathcal{G} \mid u_1 = v \text{ or } u_2 = v\}$.

When $v \in \mathcal{N}$, then $\mathsf{Neigh}_{\mathcal{G}}(v)$ is a star-shaped graph where only the central node has non-empty content. When $v \in \mathcal{V}$, then $\mathsf{Neigh}_{\mathcal{G}}(v)$ consists of all the nodes in $\mathcal{G}$ that have some key with value v, which is a common graph with no edges and a restricted function ρ.

2.3 Value Types

We assume an enumerable set of *value types* $\mathcal{T}$. The reader should think of value types as `integer`, `boolean`, `date`, *etc.* Formally, for each value type $\mathtt{v} \in \mathcal{T}$, we assume that there is a set $[\![\mathtt{v}]\!] \subseteq \mathcal{V}$ of all values of that type and that each value $v \in \mathcal{V}$ belongs to some type, *i.e.*, there is at least one $\mathtt{v} \in \mathcal{T}$ such that $v \in [\![\mathtt{v}]\!]$. Finally, we assume that there is a type $\mathtt{any} \in \mathcal{T}$ such that $[\![\mathtt{any}]\!] = \mathcal{V}$.

2.4 Shapes and Schemas

We formulate all three schema languages using *shapes*, which are unary formulas describing the graph's structure around a *focus* node or a value. Shapes will be expressed in different formalisms, specific to the schema language; for each of these formalisms we will define when a focus node or value $v \in \mathcal{N} \cup \mathcal{V}$ *satisfies* shape φ in a common graph $\mathcal{G}$, written $\mathcal{G}, v \models \varphi$.

Inspired by ShEx *shape maps*, we abstract a schema $\mathcal{S}$ as a set of pairs (sel, φ), where φ is a shape and *sel* is a *selector*. A selector is also a shape, but usually a very simple one, typically checking the presence of an incident edge with a given predicate, or a property with a given key. A graph $\mathcal{G}$ is *valid w.r.t.* $\mathcal{S}$, in symbols $\mathcal{G} \models \mathcal{S}$, if

$$\mathcal{G}, v \models \mathit{sel} \quad \text{implies} \quad \mathcal{G}, v \models \varphi,$$

for all $v \in \mathcal{N} \cup \mathcal{V}$ and $(\mathit{sel}, \varphi) \in \mathcal{S}$. That is, for each focus node or value satisfying the selector, the graph around it looks as specified by the shape. We call schemas $\mathcal{S}$ and $\mathcal{S}'$ *equivalent* if $\mathcal{G} \models \mathcal{S}$ *iff* $\mathcal{G} \models \mathcal{S}'$, for all $\mathcal{G}$. In what follows, we may use $\mathit{sel} \Rightarrow \varphi$ to indicate a pair (sel, φ) from a schema $\mathcal{S}$.

Example 2. We next describe some constraints one may want to express in the domain of Example 1.

(C1) We may want the values associated to certain keys to belong to concrete datatypes, like strings or Boolean values. In our example, we want to state that the value of the key *card* is always an integer.

(C2) We may expect the existence of a value associated to a key, an outgoing edge, or even a complex path for a given source node. For our example, we require that all owners of an account have an email address defined.

(C3) We may want to express database-like uniqueness constraints. For instance, we may wish to ensure that the email address of an account owner uniquely identifies them.

Table 1: Evaluation of a path expressions.

π	$[\![\pi]\!]^{\mathcal{G}} \subseteq (\mathcal{N} \cup \mathcal{V}) \times (\mathcal{N} \times \mathcal{V})$
id	$\{(v, v) \mid v \in \mathcal{N} \cup \mathcal{V}\}$
q	$\{(v, u) \mid (v, q, u) \in \mathcal{G}\}$
π^-	$\{(v, u) \mid (u, v) \in [\![\pi]\!]^{\mathcal{G}}\}$
$\pi \cdot \pi'$	$\{(v, u) \mid \exists v' : (v, v') \in [\![\pi]\!]^{\mathcal{G}} \wedge (v', u) \in [\![\pi']\!]^{\mathcal{G}}\}$
$\pi \cup \pi'$	$[\![\pi]\!]^{\mathcal{G}} \cup [\![\pi']\!]^{\mathcal{G}}$
π^*	$[\![\mathsf{id}]\!]^{\mathcal{G}} \cup [\![\pi]\!]^{\mathcal{G}} \cup [\![\pi \cdot \pi]\!]^{\mathcal{G}} \cup \ldots$

(C4) We may want to ensure that all paths of a certain kind end in nodes with some desired properties. For example, if an account is privileged, then all users that have access to it should also be privileged.

(C5) We may want to put an upper bound on the number of nodes reached from a given node by certain paths. For instance, every user may have access to at most 5 accounts.

3 SHACL on common graphs

We first treat SHACL, because it is conceptually the simplest of the three languages. It is essentially a logic—some call it a *description logic in disguise* [7]. Our abstraction is inspired by [25]. We focus on the standard, non-recursive SHACL, leaving recursive extensions [2, 6, 14, 38, 40] for the future. Some features of SHACL are incompatible with common graphs, and are therefore omitted (see [1, Appendix B]).

Definition 3 (Path Expression). *A* path expression π *is given by the following grammar:*

$$\pi ::= \mathsf{id} \mid q \mid \pi^- \mid \pi \cdot \pi \mid \pi \cup \pi \mid \pi^* .$$

with $q \in \mathcal{P} \cup \mathcal{K}$ *and* id *the identity relation (or empty word).*

Definition 4 (SHACL Shape). *A SHACL shape* φ *is given by the following grammar:*

$$\varphi ::= \top \mid \mathsf{test}(c) \mid \mathsf{test}(\mathbb{v}) \mid \mathsf{closed}(Q) \mid \mathsf{eq}(\pi, p) \mid \mathsf{disj}(\pi, p) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \exists^{\geq n}\pi.\varphi \mid \exists^{\leq n}\pi.\varphi .$$

with $c \in \mathcal{V}$, $\mathbb{v} \in \mathcal{T}$, $Q \subseteq_{fin} \mathcal{P} \cup \mathcal{K}$, $p \in \mathcal{P}$, *and* n *a natural number. We may use* $\exists\pi.\varphi$ *as syntactic sugar for* $\exists^{\geq 1}\pi.\varphi$.

Definition 5 (SHACL Selector). *A SHACL selector sel is a SHACL shape of a restricted form, given by the following grammar:*

$$sel ::= \exists q.\top \mid \exists q^-.\top \mid \mathsf{test}(c) .$$

with $q \in \mathcal{P} \cup \mathcal{K}$, *and* $c \in \mathcal{V}$.

Putting it together, a *SHACL Schema* $\mathcal{S}$ is a finite set of pairs (sel, φ), where *sel* is a SHACL selector and φ is a SHACL shape.

To define the semantics of SHACL schemas, we first define in Table 1 the semantics of a SHACL path expression π on a graph $\mathcal{G}$ as a binary relation $[\![\pi]\!]^{\mathcal{G}}$ over $\mathcal{N} \cup \mathcal{V}$. The semantics of SHACL shapes is defined in Table 2, which specifies when a node or value v *satisfies* a SHACL shape φ *w.r.t.* a $\mathcal{G}$, written $\mathcal{G}, v \vDash \varphi$. Note that both $[\![\pi]\!]^{\mathcal{G}}$ and $\{v \in \mathcal{N} \cup \mathcal{V} \mid \mathcal{G}, v \vDash \varphi\}$ may be infinite: for example, $[\![\mathsf{id}]\!]^{\mathcal{G}}$ is the identity relation over the infinite set $\mathcal{N} \cup \mathcal{V}$.

Table 2: Semantics of a SHACL shape φ .

φ	$\mathcal{G}, v \vDash \varphi$ if:
$\top$	trivially satisfied
$\mathsf{test}(c)$	$v = c$
$\mathsf{test}(\mathbb{v})$	$v \in [\![\mathbb{v}]\!]$
$\mathsf{closed}(Q)$	$\forall p \in (\mathcal{P} \cup \mathcal{K}) \setminus Q : \text{not } \mathcal{G}, v \vDash \exists^{\geq 1} p.\top$
$\mathsf{eq}(\pi, p)$	$\{u \mid (v, u) \in [\![\pi]\!]^{\mathcal{G}}\} = \{u \mid (v, u) \in [\![p]\!]^{\mathcal{G}}\}$
$\mathsf{disj}(\pi, p)$	$\{u \mid (v, u) \in [\![\pi]\!]^{\mathcal{G}}\} \cap \{u \mid (v, u) \in [\![p]\!]^{\mathcal{G}}\} = \emptyset$
$\neg\varphi$	not $\mathcal{G}, v \vDash \varphi$
$\varphi \wedge \varphi'$	$\mathcal{G}, v \vDash \varphi$ and $\mathcal{G}, v \vDash \varphi'$
$\varphi \vee \varphi'$	$\mathcal{G}, v \vDash \varphi$ or $\mathcal{G}, v \vDash \varphi'$
$\exists^{\geq n}\pi.\varphi$	$\#\{u \mid (v, u) \in [\![\pi]\!]^{\mathcal{G}} \wedge \mathcal{G}, u \vDash \varphi\} \geq n$
$\exists^{\leq n}\pi.\varphi$	$\#\{u \mid (v, u) \in [\![\pi]\!]^{\mathcal{G}} \wedge \mathcal{G}, u \vDash \varphi\} \leq n$

The semantics of SHACL schemas then follows Section 2.4. Importantly, SHACL selectors always select a finite subset of $\mathcal{N} \cup \mathcal{V}$: the selected nodes or values come either from the selector itself, in the case of $\mathsf{test}(c)$, or from $\mathcal{G}$, in the remaining four cases. For example, $\exists p.\top$ selects those nodes of $\mathcal{G}$ that have an outgoing p-edge in $\mathcal{G}$—it is grounded to $\mathcal{G}$ in the second line of Table 1. In consequence, each pair (sel, φ) in a SHACL schema tests the inclusion of a finite set of nodes or values in a possibly infinite set.

Example 3. For better readability we write $\exists\pi$ instead of $\exists^{\geq 1}\pi.\top$ (that is, we omit $\top$) and $\forall\pi.\varphi$ instead of $\exists^{\leq 0}\pi.\neg\varphi$. Let us see how the constraints from Example 2 can be handled in SHACL. For (C1), we assume the value type $\mathtt{int}$ with the obvious meaning. The following SHACL constraints express the constraints (C1–C5):

$$\exists card^- \Rightarrow \mathsf{test}(\mathtt{int}) \qquad \text{(C1)}$$

$$\exists \mathsf{ownsAccount} \Rightarrow \exists email \qquad \text{(C2)}$$

$$\exists email^- \Rightarrow \exists^{\leq 1} email^- \qquad \text{(C3)}$$

$$\exists card \Rightarrow (\exists privileged.\, \neg\mathsf{test}(true)) \vee \forall \mathsf{hasAcccess}^-.\, (\exists privileged.\, \mathsf{test}(true)) \qquad \text{(C4)}$$

$$\exists email \Rightarrow \exists^{\leq 5} \mathsf{hasAcccess}. \qquad \text{(C5)}$$

Concerning constraint (C3), notice that by using inverse *email* edges, the constraint indeed states that the email addresses uniquely identify users.

The constructs $\mathsf{eq}(\pi, p)$ and $\mathsf{disj}(\pi, p)$ are unique to SHACL. Let us see them in use.

Example 4. Using $\mathsf{eq}(\pi, p)$, we can say, for instance, that an owner of an account also has access to it:

$$\exists \mathsf{ownsAccount} \Rightarrow \mathsf{eq}(\mathsf{hasAcccess} \cup \mathsf{ownsAccount}, \mathsf{hasAcccess}) .$$

Note how we use eq and $\cup$ to express that the existence of one path (ownsAccount) implies the existence of another path (hasAcccess) with the same endpoints.

A key feature in SHACL that is not available in ShEx is the ability to use regular expressions to talk about complex paths. This provides a limited, still non-trivial, form of recursive navigation

in the graph, even though the standard SHACL does not support recursive constraints (in contrast to standard ShEx).

Example 5. Suppose that in Figure 1, we impose that for every node with a *privileged* key, either its value is *false* or, along inverse invited edges there is a unique, privileged "ancestor", which has no further inverse invited edges. This is expressible as follows:

$$\exists \mathit{privileged} \Rightarrow \exists \mathit{privileged}.\,\mathsf{test}(\mathit{false}) \vee \exists^{\leq 1}\mathsf{invited}^{-*}.\,(\exists \mathit{privileged}.\,\mathsf{test}(\mathit{true}) \wedge \exists^{\leq 0}\mathsf{invited}^{-})\,.$$

4 ShEx on common graphs

While SHACL is conceptually the simplest of the three languages, ShEx lies at the opposite end of the spectrum. It is an intricate, nested combination of a simple logic for shapes and a powerful formalism (triple expressions) for generating the allowed neighbourhoods. In this work we focus on non-recursive ShEx, where shapes and triple expressions can be nested multiple times, but cannot be recursive. This allows us to simplify the abstraction without compromising our primary goal of understanding the common features, as neither PG-Schema nor standard SHACL support such a general recursion mechanism. The abstraction of ShEx over common graphs is based on the treatment of ShEx on RDF triples [9]. Deviations from standard ShEx are discussed in [1, Appendix C].

Definition 6 (shapes and triple expressions). *ShEx* shapes φ *and* closed triple expressions e *are defined by the grammar*

$$\varphi ::= \mathsf{test}(c) \mid \mathsf{test}(\mathbb{v}) \mid \{e\,;\mathit{op}_-\} \mid \{e\,;\mathit{op}_\pm\} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi\,.$$

$$e ::= \varepsilon \mid q.\varphi \mid q^-.\varphi \mid e\,;e \mid e \mid e \mid e^*\,.$$

$$\mathit{op}_- ::= (\neg R^-)^*\,.$$

$$\mathit{op}_\pm ::= (\neg R^-)^*;(\neg Q)^*\,.$$

where $c \in \mathcal{V}$, $\mathbb{v} \in \mathcal{T}$, $q \in \mathcal{P} \cup \mathcal{K}$, *and* $R, Q \subseteq_{\mathit{fin}} \mathcal{P} \cup \mathcal{K}$. *We refer to expressions derived from* $e\,;\mathit{op}_-$ *and* $e\,;\mathit{op}_\pm$ *as* half-open *and* open triple expressions, *respectively.*

The notion of satisfaction for ShEx shapes and the semantics of triple expressions are defined by mutual recursion in Table 3 and Table 4. Triple expressions are used to specify neighbourhoods of nodes and values. They require to consider incoming and outgoing edges separately. For this purpose we decorate incoming edges with $^-$. Formally, we introduce a fresh predicate p^- for each $p \in \mathcal{P}$ and a fresh key k^- for each $k \in \mathcal{K}$. We let $\mathcal{P}^- = \{p^- \mid p \in \mathcal{P}\}$, $\mathcal{K}^- = \{k^- \mid k \in \mathcal{K}\}$, $\mathcal{E}^- = \mathcal{N} \times \mathcal{P}^- \times \mathcal{N} \cup \mathcal{V} \times \mathcal{K}^- \times \mathcal{N}$, and define $\mathsf{Neigh}^{\pm}_{\mathcal{G}}(v) \subseteq \mathcal{E} \cup \mathcal{E}^-$ as

$$\{(v,p,v') \mid (v,p,v') \in \mathcal{G}\} \cup \{(v,p^-,v') \mid (v',p,v) \in \mathcal{G}\}\,.$$

Compared to $\mathsf{Neigh}_{\mathcal{G}}(v)$, apart from flipping the incoming edges and marking them with $^-$, we also represent each loop (v,p,v) twice: once as an outgoing edge (v,p,v) and once as an incoming edge (v,p^-,v). In Table 4, we treat $\neg Q$ and $\neg R^-$ as triple expressions. So, the rule for e^* gives semantics to $(\neg Q)^*$ and $(\neg R^-)^*$, and the rule for $e_1\,;e_2$ gives semantics to open and half-open triple expressions. In Table 3, f is an open or half-open triple expression.

Closed triple expressions e define neighbourhoods that use only a finite number of predicates and keys (also called *closed* in ShEx terminology) and cannot be directly used in shape expressions. Half-open triple expressions $e\,;(\neg R^-)^*$ allow any *incoming* triples whose predicate or key is not in R. Open triple expressions $e\,;(\neg R^-)^*\,;(\neg Q)^*$ additionally allow any *outgoing* triples whose predicate or key is not in Q. Let $\top = \varepsilon\,;(\neg\emptyset^-)^*\,;(\neg\emptyset)^*$. Then $\top$ describes all possible neighbourhoods, and $\{\top\}$ is satisfied in every node and in every value of every graph.

Table 3: Satisfaction of ShEx shapes.

φ	$\mathcal{G}, v \vDash \varphi$ for $v \in \mathcal{N} \cup \mathcal{V}$
$\mathsf{test}(c)$	$v = c$
$\mathsf{test}(\mathbb{v})$	$v \in [\![\mathbb{v}]\!]$
$\{f\}$	$\mathsf{Neigh}^{\pm}_{\mathcal{G}}(v) \in [\![f]\!]^{\mathcal{G}}_v$
$\varphi_1 \wedge \varphi_2$	$\mathcal{G}, v \vDash \varphi_1$ and $\mathcal{G}, v \vDash \varphi_2$
$\varphi_1 \vee \varphi_2$	$\mathcal{G}, v \vDash \varphi_1$ or $\mathcal{G}, v \vDash \varphi_2$
$\neg\varphi$	not $\mathcal{G}, v \vDash \varphi$

Example 6. The ShEx shape $\{p.\varphi_1\,;p.\varphi_2;\top\}$ specifies nodes with at least two different p-successors, one satisfying φ_1 and one satisfying φ_2. Note that this is different from SHACL shape $\exists p.\varphi_1 \wedge \exists p.\varphi_2$ which says that the node has a p-successor satisfying φ_1 and a p-successor satisfying φ_2, but they might not be different.

Example 7. Assume that integers and strings are represented by $\mathtt{int}, \mathtt{str} \in \mathcal{T}$. The ShEx shape

$$\{\mathit{email}.\mathsf{test}(\mathtt{str})\ ;(\mathit{card}.\mathsf{test}(\mathtt{int}) \mid \varepsilon)\ ;(\neg\emptyset^-)^*\}$$

specifies nodes with an *email* property with a string value, an optional *card* property with an integer value, arbitrary incoming edges, and no other properties or outgoing edges. To allow additional properties and outgoing edges, we replace $(\neg\emptyset^-)^*$ with $\top$. The modified shape can be rewritten using $\wedge$ as

$$\{\mathit{email}.\mathsf{test}(\mathtt{str})\ ;\top\} \wedge \{(\mathit{card}.\mathsf{test}(\mathtt{int}) \mid \varepsilon)\ ;\top\}$$

but the original shape cannot be rewritten in a similar way.

Definition 7 (ShEx Selectors). *A ShEx selector is a ShEx shape of a restricted form, defined by the grammar*

$$\mathit{sel} ::= \mathsf{test}(c) \mid \{q.\mathsf{test}(c)\,;\top\} \mid \{q.\{\top\}\,;\top\} \mid \{q^-.\{\top\}\,;\top\}\,.$$

where $q \in \mathcal{P} \cup \mathcal{K}$ *and* $c \in \mathcal{V}$.

Following Section 2.4, a *ShEx schema* $\mathcal{S}$ is a set of pairs of the form (sel, φ) where φ is a ShEx shape and sel is a ShEx selector.

In what follows, for a positive integer n, we write e^n for $e\,;\ldots;e$ where e is repeated n times, $e^{\leq n}$ for $\varepsilon \mid e^1 \mid \ldots \mid e^n$, and $e^{\geq n}$ for $e^n\,;e^*$. For a closed triple expression e, we let $\{e\}^\circ = \{e\,;(\neg R^-)^*\,;(\neg Q)^*\}$ where Q is the set of predicates and keys that appear *directly* in e (as opposed to appearing in φ for a sub-expression $q.\varphi$ of e) and R is the set of predicates and keys whose inversions appear directly in e. For instance, if $e = p.\{q.\{\top\}\,;p^-.\{\top\}\}$, then $Q = \{p\}$ and $R = \emptyset$.

Table 4: Semantics of triple expressions.

e	$[\![e]\!]^{\mathcal{G}}_v \subseteq 2^{\mathcal{E}\cup\mathcal{E}^-}$
ε	$\{\emptyset\}$
$q.\varphi$	$\{\{(v,q,v')\} \subseteq \mathcal{E} \mid \mathcal{G}, v' \vDash \varphi\}$
$q^-.\varphi$	$\{\{(v,q^-,v')\} \subseteq \mathcal{E}^- \mid \mathcal{G}, v' \vDash \varphi\}$
$e_1\,;e_2$	$\left\{T_1 \cup T_2 \;\middle|\; T_1 \in [\![e_1]\!]^{\mathcal{G}}_v,\ T_2 \in [\![e_2]\!]^{\mathcal{G}}_v,\ T_1 \cap T_2 = \emptyset\right\}$
$e_1 \mid e_2$	$[\![e_1]\!]^{\mathcal{G}}_v \cup [\![e_2]\!]^{\mathcal{G}}_v$
e^*	$\{\emptyset\} \cup \bigcup_{n=1}^{\infty} \left\{ T_1 \cup \cdots \cup T_n \;\middle|\; \begin{array}{l} T_1,\ldots,T_n \in [\![e]\!]^{\mathcal{G}}_v \text{ and} \\ T_i \cap T_j = \emptyset \text{ for all } i \neq j \end{array} \right\}$
$\neg Q$	$\{\{(v,q,v')\} \subseteq \mathcal{E} \mid q \notin Q\}$
$\neg R^-$	$\{\{(v,q^-,v')\} \subseteq \mathcal{E}^- \mid q \notin R\}$

Example 8. Let us now see how the concrete constraints from Example 2 can be handled in ShEx.

$$\{card^-.\{\top\};\top\} \Rightarrow \mathsf{test}(\mathtt{int}) \tag{C1}$$

$$\{\mathsf{ownsAccount}.\{\top\};\top\} \Rightarrow \{email.\{\top\};\top\} \tag{C2}$$

$$\{email^-.\{\top\};\top\} \Rightarrow \left\{(email^-.\{\top\})^{\leq 1}\right\}^{\circ} \tag{C3}$$

$$\{card.\{\top\};\top\} \Rightarrow \{privileged.\neg\mathsf{test}(true)\}^{\circ} \vee \left\{(\mathsf{hasAcccess}^-.\{privileged.\mathsf{test}(true)\}^{\circ})^*\right\}^{\circ} \tag{C4}$$

$$\{email.\{\top\};\top\} \Rightarrow \left\{(\mathsf{hasAcccess}.\{\top\})^{\leq 5}\right\}^{\circ} \tag{C5}$$

We next show a more complex example, which illustrates the power of ShEx that is not readily available in SHACL or PG-Schema.

Example 9. Suppose that we want to express the following constraint on each user who owns an account: the number of accounts to which the user has access is greater or equal to the number of accounts that the user owns. We can do this in ShEx as follows:

$$\{\mathsf{ownsAccount}.\{\top\};\top\} \Rightarrow$$

$$\left\{(\mathsf{hasAcccess}.\{\top\})^*;(\mathsf{ownsAccount}.\{\top\};\mathsf{hasAcccess}.\{\top\})^*\right\}^{\circ}$$

Similarly to the above (yet more abstractly) consider the following requirement: for the node c, the number of outgoing p-edges is *equal* to the number of outgoing q-edges. This can be expressed in ShEx using $\mathsf{test}(c) \Rightarrow \{(p.\{\top\};q.\{\top\})^*\}^{\circ}$ but cannot be expressed in SHACL (see [1, Appendix C]).

Finally, let us see why ShEx and SHACL count differently.

Example 10. The following SHACL schema ensures that from every node with an outgoing hasAcccess-edge, exactly two nodes are accessible via a hasAcccess-edge or an ownsAccount-edge:

$$\exists \mathsf{hasAcccess} \Rightarrow \exists^{=2}(\mathsf{hasAcccess} \cup \mathsf{ownsAccount}).\top$$

Here $\exists^{=n}\pi.\varphi$ is a shorthand for $\exists^{\leq n}\pi.\varphi \wedge \exists^{\geq n}\pi.\varphi$. For instance, in Figure 2, the graph on the right is valid, whereas the one on the left is not. The same constraint cannot be expressed in ShEx because ShEx cannot distinguish these two graphs (see [1, Appendix C]). The reason is that ShEx triple expressions count triples adjacent to a node, whereas SHACL and PG-Schema count nodes on the opposite end of such triples. This makes counting edges simpler in ShEx: the ShEx shape $\left\{(p.\{\top\} \mid q.\{\top\})^2;(\neg\emptyset^-)^*\right\}$ allows exactly

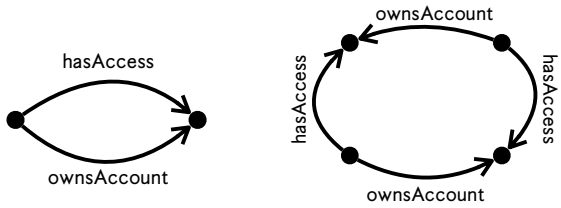


Figure 2: Two graphs indistinguishable by ShEx

Table 5: Semantics of content types.

$\mathbb{c}$	$[\![\mathbb{c}]\!] \subseteq \mathcal{R}$
$[\![\top]\!]$	$\mathcal{R}$
$[\![\{\}]\!]$	$\{\mathbf{r}_\emptyset\}$
$[\![\{k:\mathbb{v}\}]\!]$	$\{\{(k,w)\} \mid w \in [\![\mathbb{v}]\!]\}$
$[\![\,\mathbb{c}_1 \,\&\, \mathbb{c}_2\,]\!]$	$\{(r_1 \cup r_2) \in \mathcal{R} \mid r_1 \in [\![\mathbb{c}_1]\!] \wedge r_2 \in [\![\mathbb{c}_2]\!]\}$
$[\![\,\mathbb{c}_1 \mid \mathbb{c}_2\,]\!]$	$[\![\mathbb{c}_1]\!] \cup [\![\mathbb{c}_2]\!]$

two outgoing edges labelled p or q. In SHACL this is written as $(\exists^{=2}p.\top \wedge \exists^{=0}q.\top) \vee (\exists^{=2}q.\top \wedge \exists^{=0}p.\top) \vee (\exists^{=1}p.\top \wedge \exists^{=1}q.\top)$.

5 Shape-based PG-Schema

Shape-based PG-Schema is a non-recursive combination of a logic and two generative formalisms. It uses path expressions to specify paths (as in SHACL), and *content types* to specify node contents. Both path expressions and content types are then used in formulas defining shapes. Content types in PG-Schema play a role similar to triple expressions in ShEx, but they are only used for properties. Because all properties of a node must have different keys, they are much simpler than triple expressions (in fact, they can be translated into a fragment of SHACL). Unlike for SHACL and ShEx, the abstraction of shape-based PG-Schema departs significantly from the original design. Original PG-Schema uses queries written in an external query language, which is left unspecified aside from some basic assumptions about the expressive power. Here we use a specific query language (PG-path expressions). Importantly, up to the choice of the query language, the abstraction we present here faithfully captures the expressive power of the original PG-Schema. A detailed comparison can be found in [1, Appendix D].

Definition 8 (Content type). *A* content type *is an expression* $\mathbb{c}$ *of the form defined by the grammar*

$$\mathbb{c} ::= \top \mid \{\} \mid \{k:\mathbb{v}\} \mid \mathbb{c}\,\&\,\mathbb{c} \mid \mathbb{c} \mid \mathbb{c}\,.$$

where $k \in \mathcal{K}$ *and* $\mathbb{v} \in \mathcal{T}$.

Recall that $\mathcal{R}$ is the set of all records. We write $\mathbf{r}_\emptyset$ for the empty record. The semantics of content types is defined in Table 5. Note that $[\![\mathbb{c}]\!]$ is independent from $\mathcal{G}$ and can be infinite.

Example 11. We assume integers and strings are represented via $\mathtt{int}, \mathtt{str} \in \mathcal{T}$. Suppose we want to create a content type for nodes that have a string value for the *email* key and *optionally* have an

Table 6: Semantics of PG-path expressions.

π	$[\![\pi]\!]^{\mathcal{G}} \subseteq (\mathcal{N} \cup \mathcal{V}) \times (\mathcal{N} \cup \mathcal{V})$ for $\mathcal{G} = (E, \rho)$
$[k = c]$	$\{(u,u) \mid u \in \mathsf{Nodes}(\mathcal{G}) \wedge (k,c) \in \rho(u)\}$
$\neg[k = c]$	$\{(u,u) \mid u \in \mathsf{Nodes}(\mathcal{G}) \wedge (k,c) \notin \rho(u)\}$
$\mathbb{c}$	$\{(u,u) \mid u \in \mathsf{Nodes}(\mathcal{G}) \wedge \rho(u) \in [\![\mathbb{c}]\!]\}$
$\neg\mathbb{c}$	$\{(u,u) \mid u \in \mathsf{Nodes}(\mathcal{G}) \wedge \rho(u) \notin [\![\mathbb{c}]\!]\}$
k	$\{(u,w) \mid \rho(u,k) = w\}$
p	$\{(u,v) \mid (u,p,v) \in E\}$
$\neg P$	$\{(u,v) \mid \exists p : (u,p,v) \in E \wedge p \notin P\}$
π^-	$\{(u,v) \mid (v,u) \in [\![\pi]\!]^{\mathcal{G}}\}$
$\pi \cdot \pi'$	$\{(u,v) \mid \exists w : (u,w) \in [\![\pi]\!]^{\mathcal{G}} \wedge (w,v) \in [\![\pi']\!]^{\mathcal{G}}\}$
$\pi \cup \pi'$	$[\![\pi]\!]^{\mathcal{G}} \cup [\![\pi']\!]^{\mathcal{G}}$
π^*	$\{(u,u) \mid u \in \mathsf{Nodes}(\mathcal{G})\} \cup [\![\pi]\!]^{\mathcal{G}} \cup [\![\pi \cdot \pi]\!]^{\mathcal{G}} \cup \ldots$

integer value for the *card* key. No other key-value pairs are allowed. We should then use $\{email : \mathtt{str}\}$ & $(\{card : \mathtt{int}\} \mid \{\})$.

Definition 9 (PG-path expressions). *A PG-path expression is an expression π of the form defined by the grammar*

$$\pi ::= \bar{\pi} \mid \bar{\pi} \cdot k \mid k^- \cdot \bar{\pi} \mid k^- \cdot \bar{\pi} \cdot k' \,.$$

$$\bar{\pi} ::= [k = c] \mid \neg[k = c] \mid \mathbb{c} \mid \neg\mathbb{c} \mid p \mid \neg P \mid \bar{\pi}^- \mid \bar{\pi} \cdot \bar{\pi} \mid \bar{\pi} \cup \bar{\pi} \mid \bar{\pi}^* \,.$$

where $k, k' \in \mathcal{K}$, $c \in \mathcal{V}$, $\mathbb{c}$ is a content type, $p \in \mathcal{P}$, and $P \subseteq_{fin} \mathcal{P}$. We use k, k^-, and $k^- \cdot k'$ as short-hands for PG-path expressions $\top \cdot k$, $k^- \cdot \top$, and $k^- \cdot \top \cdot k'$, respectively.

Unlike in SHACL, PG-path expressions cannot navigate freely through values. In property graphs this would correspond to a join, which is a costly operation. Indeed, existing query languages for property graphs do not allow joins under *. However, PG-path expressions may start in a value and finish in a value. This leads to *node-to-node*, *node-to-value*, *value-to-node*, and *value-to-value* expressions, reflected in the four cases in the first rule of the grammar.

The semantics of PG-path expression π for graph $\mathcal{G}$ is a binary relation over $\mathsf{Nodes}(\mathcal{G}) \cup \mathsf{Values}(\mathcal{G})$, defined in Table 6. In the table, k is treated as any other subexpressions, eventhough it can only be used at the end of a PG-path expression, or in the beginning as k^-. Notice that $\neg\mathbb{c}$ matches nodes whose content is not of type $\mathbb{c}$, $\neg P$ matches edges with a label that is not in P (in particular, $\neg\emptyset$ matches all edges). Also, $[\![\pi]\!]^{\mathcal{G}}$ is always a subset of $\mathcal{N} \times \mathcal{N}$, $\mathcal{N} \times \mathcal{V}$, $\mathcal{V} \times \mathcal{N}$, or $\mathcal{V} \times \mathcal{V}$, corresponding to the four kinds of PG-path expressions discussed above.

Definition 10 (PG-Shapes). *A PG-Shape is an expression φ defined by the following grammar:*

$$\varphi ::= \exists^{\leq n} \pi \mid \exists^{\geq n} \pi \mid \varphi \wedge \varphi \,.$$

where π is a PG-path expression. We write $\exists$ and $\nexists$ for $\exists^{\geq 1}$ and $\exists^{\leq 0}$.

The semantics of PG-shapes is defined in Table 7. We say $v \in \mathcal{N} \cup \mathcal{V}$ *satisfies* a PG-shape φ in a graph $\mathcal{G}$ if $\mathcal{G}, v \vDash \varphi$. Every PG-shape is satisfied by nodes only or by values only.

Definition 11 (PG-Selectors). *A* PG-selector *is a PG-shape of the form $\exists\,\pi$.*

Table 7: Satisfaction of PG-shapes

φ	$\mathcal{G}, v \vDash \varphi$ for $v \in \mathcal{N} \cup \mathcal{V}$
$\exists^{\leq n}\,\pi$	$\#\{v' \mid (v,v') \in [\![\pi]\!]^{\mathcal{G}}\} \leq n$
$\exists^{\geq n}\,\pi$	$\#\{v' \mid (v,v') \in [\![\pi]\!]^{\mathcal{G}}\} \geq n$
$\varphi_1 \wedge \varphi_2$	$\mathcal{G}, v \vDash \varphi_1$ and $\mathcal{G}, v \vDash \varphi_2$

A *PG-Schema* $\mathcal{S}$ is a finite set of pairs (sel, φ) where *sel* is a PG-selector and φ is a PG-shape. The semantics of PG-Schemas is defned just like in Section 2.4.

Example 12. The constraints (C1-C5) from Example 2 can be handled in PG-Schema as follows:

$$\exists card \Rightarrow \exists(\{card : \mathtt{int}\} \,\&\, \top) \tag{C1}$$

$$\exists \mathsf{ownsAccount} \Rightarrow \exists email \tag{C2}$$

$$\exists email^- \Rightarrow \exists^{\leq 1} email^- \tag{C3}$$

$$\exists\,(\{card : \mathtt{any}\} \,\&\, \top) \cdot \{privileged : true\} \Rightarrow \nexists\, \mathsf{hasAcccess}^- \cdot \neg\{privileged : true\} \tag{C4}$$

$$\exists email \Rightarrow \exists^{\leq 5}\, \mathsf{hasAcccess} \tag{C5}$$

Notice that in rule (C1), we indeed need $\exists card$, rather than $\exists card^-$, because there is no PG-Shape to state that the selected value is of type $\mathtt{int}$, and so we formulate C1 as a statement about nodes.

A characteristic feature of PG-Schema, revealing its database provenience, is that it can close the whole graph by imposing restrictions on all nodes.

Example 13. Given a common graph such as the one in Figure 1, we might want to express that each node has a key *privileged* with a boolean value and either a key *card* with an integer value or a key *email* with a string value, and no other keys are allowed. In PG-Schema this can be expressed as follows:

$$\exists\top \Rightarrow \exists\{privileged : \mathtt{bool}\} \,\&\, (\{card : \mathtt{int}\} \mid \{email : \mathtt{str}\})\,.$$

We can also forbid all predicates except those mentioned in the running example: $\exists\top \Rightarrow \nexists\neg\{\mathsf{ownsAccount}, \mathsf{hasAcccess}, \mathsf{invited}\}$.

6 Common Graph Schema Language

We now present the Common Graph Schema Language (CoGSL), which combines the core functionalities shared by SHACL, ShEx, and PG-Schema (over common graphs).

Let us begin by examining the restrictions that need to be imposed. We shall refer to shapes and selectors used in CoGSL as *common shapes* and *common selectors*. Common shapes cannot be closed under disjunction and negation, because PG-Schema shapes are purely conjunctive. For the same reason common shapes cannot be nested. Kleene star * cannot be allowed in path expressions because we consider ShEx without recursion. Supporting path expressions traversing more than one edge under counting quantifiers is impossible as this is not expressible in ShEx. Supporting disjunctions of labels of the form $p_1 \cup p_2$ is also impossible, due to a mismatch in the approach to counting: while SHACL and PG-Schema count nodes and values, ShEx counts triples, as illustrated in Example 10.

Closed content types and $\neg P$ cannot be used freely, because neither SHACL nor ShEx are capable of closing only properties or only predicate edges: both must be closed at the same time.

Finally, selectors are restricted because SHACL and ShEx do not support $\top$ as a selector; that is, one cannot say that each node (or value) in the graph satisfies a given shape. This means that SHACL and ShEx schemas always allow a disconnected part of the graph that uses only predicates and keys not mentioned in the schema, whereas PG-Schema can disallow it (see Example 13).

Putting these restrictions together we obtain the Common Graph Schema Language. We define it below as a fragment of PG-Schema.

Definition 12 (common shape). *A* common shape φ *is an expression given by the grammar*

$$\begin{aligned}
\varphi &::= \exists\, \pi \mid \exists^{\leq n}\, \pi_1 \mid \exists^{\geq n}\, \pi_1 \mid \exists\, \mathbb{c} \wedge \nexists \neg P \mid \varphi \wedge \varphi \ .\\
\mathbb{c} &::= \{\} \mid \{k : \mathbb{v}\} \mid \mathbb{c}\ \&\ \mathbb{c} \mid \mathbb{c} \mid \mathbb{c} \ .\\
\pi_0 &::= [k = c] \mid \neg[k = c] \mid \mathbb{c}\ \&\ \top \mid \neg(\mathbb{c}\ \&\ \top) \mid \pi_0 \cdot \pi_0 \ .\\
\pi_1 &::= \pi_0 \cdot p \cdot \pi_0 \mid \pi_0 \cdot p^- \cdot \pi_0 \mid \pi_0 \cdot k \mid k^- \cdot \pi_0 \ .\\
\bar{\pi} &::= \pi_0 \mid p \mid \bar{\pi}^- \mid \bar{\pi} \cdot \bar{\pi} \mid \bar{\pi} \cup \bar{\pi} \ .\\
\pi &::= \bar{\pi} \mid \bar{\pi} \cdot k \mid k^- \cdot \bar{\pi} \mid k^- \cdot \bar{\pi} \cdot k' \ .
\end{aligned}$$

where $n \in \mathbb{N}$, $P \subseteq_{fin} \mathcal{P}$, $k, k' \in \mathcal{K}$, $c \in \mathcal{V}$, *and* $p \in \mathcal{P}$.

That is, $\mathbb{c}$ is a content type that does not use $\top$ (a *closed* content type), π_0 is a PG-path expression that always stays in the same node (a *filter*), π_1 is a PG-path expression that traverses a single edge or property (forward or backwards), and π is a PG-path expression that uses neither * nor $\neg P$. Moreover, π_0, π_1, and π can only use *open* content types; that is, content types of the form $\mathbb{c}\ \&\ \top$. The use of $\neg P$ is limited to closing the neighbourhood of a node (this is the only way PG-Schema can do it).

Definition 13 (common selector). *A* common selector *is a common shape of one of the following forms*

$$\exists\, k,\ \exists\, p \cdot \pi,\ \exists\, p^- \cdot \pi,\ \exists\, [k = c] \cdot \pi,\ \exists\, (\{k : \mathbb{v}\}\ \&\ \top) \cdot \pi,\ \exists\, k^- \cdot \pi,$$

where $k \in \mathcal{K}$, $p \in \mathcal{P}$, $c \in \mathcal{V}$, $\mathbb{v} \in \mathcal{T}$ *and* $\pi = \bar{\pi}$ *or* $\pi = \bar{\pi} \cdot k'$ *for some PG-path expression* $\bar{\pi}$ *generated by the grammar in Definition 12 and some* $k' \in \mathcal{K}$.

That is, a common selector is a common shape of the form $\exists\, \pi$ such that the PG-path expression π requires the focus node or value to occur in a triple with a specified predicate or key.

A *common schema* is a finite set of pairs (sel, φ) where sel is a common selector and φ is a common shape. The semantics is inherited from PG-Schema.

As we have seen, the constraints (C1)-(C5) from our running example can be expressed in all three formalisms; the PG-Schema representation from Example 12 is also a common schema.

Proposition 1. *For every common schema there exist equivalent SHACL and ShEx schemas.*

The translation is relatively straightforward (see Appendix C). The two main observations are that star-free PG-path expressions can be simulated by nested SHACL and ShEx shapes, and that closure of SHACL and ShEx shapes under Boolean connectives allows encoding complex selectors in the shape (as the antecedent of an implication). We illustrate the latter in Example 14.

Example 14 (Complex paths in selectors). We want to express that all users who have invited a user who has invited someone (so there is a path following two invited edges) must have a key *email* of type $\mathbb{str}$. In PG-schema we express this as:

$$\exists \mathsf{invited} \cdot \mathsf{invited} \Rightarrow \{email : \mathbb{str}\}\ \&\ \top$$

At first glance, it seems unclear how to express this in the other formalisms, since they do not permit paths in the selector. However, we can see that paths in selectors can be encoded into the shape: In SHACL, using the same example, we do this by

$$\exists \mathsf{invited} \Rightarrow \neg(\exists \mathsf{invited} \cdot \mathsf{invited}) \vee \exists email.\mathsf{test}(\mathbb{str})$$

And in ShEx for this example would be:

$$\{\mathsf{invited}.\ \{\top\}\,;\top\} \Rightarrow \neg\varphi_2 \vee \{email.\mathsf{test}(\mathbb{str})\}^\circ$$

where $\varphi_2 = \left\{\ (\mathsf{invited}.\varphi_1)^{\geq 1}\ \right\}^\circ$ and $\varphi_1 = \left\{\ \mathsf{invited}.\ \{\top\}^{\geq 1}\ \right\}^\circ$. That is, φ_1 is satisfied by nodes that have an outgoing path invited, and φ_2 by nodes that have an outgoing path invited $\cdot$ invited. For paths of unbounded length, it is not apparent how such a translation would proceed for ShEx schemas in the absence of recursion.

7 Conclusions

We provided a formal and comprehensive comparison of the three most prominent schema languages in the Semantic Web and Graph Database communities: SHACL, ShEx, and PG-Schema. Through painstaking discussions within our working group, we managed to (1) agree on a common data model that captures features of both RDF and Property Graphs and (2) extract, for each of the languages, a core that we mutually agree on, which we define formally. Moreover, the definitions of (the cores of) each of the schema languages on a common formal framework allows readers to maximally leverage their understanding of one schema language in order to understand the others. Furthermore, this common framework allowed us to extract the Common Graph Schema Language, which is a cleanly defined set of functionalities shared by SHACL, ShEx, and PG-Schema. This commonality can serve as a basis for future efforts in integrating or translating between the languages, promoting interoperability in applications that rely on heterogeneous data models. For example, we want to investigate recursive ShEx and more expressive query languages for PG-Schema more deeply.

Acknowledgments

This work was initiated during Dagstuhl Seminar 24102 *Shapes in Graph Data*. It was funded by the Austrian Science Fund (FWF) [10.55776/COE12] (Polleres); ANR project EQUUS ANR-19-CE48-0019, project no. 431183758 by the German Research Foundation (Martens); ANGLIRU: Applying kNowledge Graphs to research data interoperabiLIty and ReUsability, code: PID2020-117912RB from the Spanish Research Agency (Labra Gayo); European Union's Horizon Europe research and innovation program under Grant Agreement No 101136244 (TARGET) (Hose and Tomaszuk); Austrian Science Fund (FWF) and netidee SCIENCE [T1349-N], and the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201] (Ahmetaj); Poland's NCN grant 2018/30/E/ST6/00042 (Murlak); and FWF standalone project P30873 (Šimkus). Mogavero is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica.

References

[1] Shqiponja Ahmetaj, Iovka Boneva, Jan Hidders, Kaja Hose, Maxime Jakubowski, Jose Emilio Labra Gayo, Wim Martens, Fabio Mogavero, Filip Murlak, Cem Okulmus, Axel Polleres, Ognjen Savković, Mantas Šimkus, and Dominik Tomaszuk. 2025. Common Foundations for SHACL, ShEx, and PG-Schema. arXiv:2502.01295 [cs.DB] https://arxiv.org/abs/2502.01295

[2] Medina Andresel, Julien Corman, Magdalena Ortiz, Juan L. Reutter, Ognjen Savkovic, and Mantas Šimkus. 2020. Stable Model Semantics for Recursive SHACL. In *The Web Conference (WWW)*. ACM / IW3C2, 1570–1580. doi:10.1145/3366423.3380229

[3] 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 (2023). doi:10.1145/3589778

[4] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savković, Michael Schmidt, Juan Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, USA, 2423–2436. doi:10.1145/3448016.3457561

[5] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. 2020. Mapping RDF databases to property graph databases. *IEEE Access* 8 (2020), 86091–86110.

[6] Bart Bogaerts and Maxime Jakubowski. 2021. Fixpoint Semantics for Recursive SHACL. In *International Conference on Logic Programming (ICLP)*, Vol. 345. 41–47. doi:10.4204/EPTCS.345.14

[7] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. 2022. SHACL: A Description Logic in Disguise. In *Logic Programming and Nonmonotonic Reasoning (LPNMR)*. Springer, 75–88. doi:10.1007/978-3-031-15707-3_7

[8] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. 2024. Expressiveness of SHACL Features and Extensions for Full Equality and Disjointness Tests. *Logical Methods in Computer Science* Volume 20, Issue 1 (Feb. 2024). doi:10.46298/lmcs-20(1:16)2024

[9] Iovka Boneva, Jose E. Labra Gayo, and Eric G. Prud'hommeaux. 2017. Semantics and Validation of Shapes Schemas for RDF. In *International Semantic Web Conference (ISWC)*. Springer, 104–120.

[10] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs.* Morgan & Claypool Publishers. doi:10.2200/S00873ED1V01Y201808DTM051

[11] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. 2008. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* Technical Report. World Wide Web Consortium.

[12] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. 2005. Named graphs. *Journal of Web Semantics* 3, 4 (2005), 247–267. doi:10.1016/j.websem.2005.09.001

[13] Julien Corman, Fernando Florenzano, Juan L. Reutter, and Ognjen Savkovic. 2019. Validating Shacl Constraints over a SPARQL Endpoint. In *International Semantic Web Conference (ISWC)*, Vol. 11778. Springer, 145–163. doi:10.1007/978-3-030-30793-6_9

[14] Julien Corman, Juan L. Reutter, and Ognjen Savković. 2018. Semantics and Validation of Recursive SHACL. In *International Semantic Web Conference (ISWC)*. Springer, 318–336.

[15] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. *RDF 1.1 Concepts and Abstract Syntax.* W3C Recommendation. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[16] Jos De Bruijn, Rubén Lara, Axel Polleres, and Dieter Fensel. 2005. OWL DL vs. OWL Flight: Conceptual modeling and reasoning for the Semantic Web. In *International conference on World Wide Web (WWW)*. 623–632.

[17] Thomas Delva, Anastasia Dimou, Maxime Jakubowski, and Jan Van den Bussche. 2023. Data Provenance for SHACL. In *International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 285–297. doi:10.48786/edbt.2023.23

[18] Michael Dyck, Jonathan Robie, and Josh Spiegel. 2017. *XML Path Language (XPath) 3.1.* W3C Recommendation. https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

[19] Nicolas Ferranti, Jairo Francisco de Souza, Shqiponja Ahmetaj, and Axel Polleres. 2024. Formalizing and Validating Wikidata's Property Constraints using SHACL and SPARQL. *Semantic Web* (2024). doi:10.3233/SW-243611

[20] Shudi (Sandy) Gao, C. M. Sperberg-McQueen, Henry S. Thompson, Noah Mendelsohn, David Beech, and Murray Maloney. 2012. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* Technical Report. World Wide Web Consortium.

[21] Olaf Hartig. 2014. Reconciliation of RDF* and Property Graphs. *CoRR* abs/1409.3288 (2014). http://arxiv.org/abs/1409.3288

[22] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. 2015. Reifying RDF: What Works Well With Wikidata?. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*, Vol. 1457. 32–47. https://ceur-ws.org/Vol-1457/SSWS2015_paper3.pdf

[23] International Organization for Standardization. 2020. *ISO/IEC 19757-3:2020 Information technology – Document Schema Definition Languages (DSDL) – Part 3: Rule-based validation using Schematron.* Standard. International Organization for Standardization, Geneva.

[24] International Organization for Standardization. 2024. *ISO/IEC 39075:2024 Information technology – Database languages – GQL.* Standard. International Organization for Standardization, Geneva.

[25] Maxime Jakubowski. 2024. *Shapes Constraint Language: Formalization, Expressiveness, and Provenance.* Ph. D. Dissertation. Universiteit Hasselt and Vrije Universiteit Brussel.

[26] Gregg Kellogg, Pierre-Antoine Champin, Olaf Hartig, and Andy Seaborne. 2024. *RDF 1.2 Concepts and Abstract Syntax.* W3C Working Draft. W3C. https://www.w3.org/TR/2024/WD-rdf12-concepts-20240822/.

[27] Holger Knublauch, James A. Hendler, and Kingsley Idehen. 2011. *SPIN — Overview and Motivation.* Technical Report. World Wide Web Consortium.

[28] Holger Knublauch and Dimitris Kontokostas. 2017. *Shapes constraint language (SHACL).* W3C Recommendation. https://www.w3.org/TR/shacl/.

[29] Jose Emilio Labra Gayo. 2022. WShEx: A language to describe and validate Wikibase entities. In *Wikidata Workshop.* https://ceur-ws.org/Vol-3262/paper3.pdf

[30] Jose Emilio Labra Gayo. 2024. Extending Shape Expressions for different types of knowledge graphs. In *Workshop on Data Quality meets Machine Learning and Knowledge Graphs.*

[31] Jose Emilio Labra Gayo, Herminio García-González, Daniel Fernández-Alvarez, and Eric Prud'hommeaux. 2019. Challenges in RDF Validation. In *Current Trends in Semantic Web Technologies: Theory and Practice.* Springer, 121–151. doi:10.1007/978-3-030-06149-4_6

[32] Jose Emilio Labra Gayo, Eric Prud'hommeaux, Iovka Boneva, and Dimitris Kontokostas. 2017. *Validating RDF Data.* Morgan & Claypool Publishers. doi:10.2200/S00786ED1V01Y201707WBE016

[33] Ora Lassila, Michael Schmidt, Olaf Hartig, Brad Bebee, Dave Bechberger, Willem Broekema, Ankesh Khandelwal, Kelvin Lawrence, Carlos Manuel Lopez Enriquez, Ronak Sharda, et al. 2023. The OneGraph vision: Challenges of breaking the graph model lock-in 1. *Semantic Web* 14, 1 (2023), 125–134.

[34] Martin Leinberger, Philipp Seifer, Tjitze Rienstra, Ralf Lämmel, and Steffen Staab. 2020. Deciding SHACL Shape Containment through Description Logics Reasoning. In *International Semantic Web Conference (ISWC)*. Springer, 366–383.

[35] Wim Martens, Frank Neven, Matthias Niewerth, and Thomas Schwentick. 2017. BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema. *ACM Trans. Database Syst.* 42, 3 (2017), 15:1–15:42. doi:10.1145/3105960

[36] Vinh Nguyen, Olivier Bodenreider, and Amit P. Sheth. 2014. Don't like RDF reification? Making statements about statements using singleton property. In *International World Wide Web Conference (WWW)*. ACM, 759–770. doi:10.1145/2566486.2567973

[37] Mikael Nilsson. 2008. *Description Set Profiles: A constraint language for Dublin Core Application Profiles.* Technical Report. Dublin Core.

[38] Cem Okulmus and Mantas Šimkus. 2024. SHACL Validation under the Well-founded Semantics. In *Proc. of KR 2024.* doi:10.24963/KR.2024/52

[39] Paolo Pareti and George Konstantinidis. 2021. A Review of SHACL: From Data Validation to Schema Reasoning for RDF Graphs. In *Reasoning Web (RW)*. Springer, 115–144. doi:10.1007/978-3-030-95481-9_6

[40] Paolo Pareti, George Konstantinidis, and Fabio Mogavero. 2022. Satisfiability and Containment of Recursive SHACL. *JWS* 74 (2022), 100721:1–24.

[41] Paolo Pareti, George Konstantinidis, Fabio Mogavero, and Timothy J. Norman. 2020. SHACL Satisfiability and Containment. In *International Semantic Web Conference (ISWC)*. Springer, 474–493.

[42] Eric Prud'hommeaux, Iovka Boneva, Jose Emilio Labra Gayo, and Gregg Kellog. 2019. *Shape Expressions Language 2.1.* W3C Community Group Report. W3C. http://shex.io/shex-semantics/.

[43] Eric Prud'hommeaux, Jose Emilio Labra Gayo, and Harold Solbrig. 2014. Shape expressions: an RDF validation and transformation language. In *International Conference on Semantic Systems (SEM)*. ACM, 32–40. doi:10.1145/2660517.2660523

[44] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2023. Extraction of Validating Shapes from very large Knowledge Graphs. *Proc. VLDB Endow.* 16, 5 (2023), 1023–1032. doi:10.14778/3579075.3579078

[45] Arthur Ryman. 2014. *Resource Shape 2.0.* Technical Report. World Wide Web Consortium.

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

a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71. doi:10.1145/3434642

[47] Philipp Seifer, Ralf Lämmel, and Steffen Staab. 2021. ProGS: Property Graph Shapes Language. In *International Semantic Web Conference (ISWC)*. Springer International Publishing, 392–409.

[48] Evren Sirin. 2010. Data validation with OWL integrity constraints. In *International Conference on Web Reasoning and Rule Systems (RR)*. Springer, 18–22.

[49] Slawek Staworko, Iovka Boneva, Jose Emilio Labra Gayo, Samuel Hym, Eric G. Prud'hommeaux, and Harold R. Solbrig. 2015. Complexity and Expressiveness of ShEx for RDF. In *International Conference on Database Theory (ICDT)*. 195–211. doi:10.4230/LIPIcs.ICDT.2015.195

[50] Dominik Tomaszuk. 2017. RDF validation: A brief survey. In *International Conference Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation (BDAS)*. Springer, 344–355.

[51] Domagoj Vrgoč, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil-Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2023. MillenniumDB: An Open-Source Graph Database System. *Data Intelligence* (06 2023), 1–39. doi:10.1162/dint_a_00209

[52] W3C OWL Working Group. 2012. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. W3C Recommendation. https://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

A Related Work

SHACL literature. The authoritative source for SHACL is the W3C recommendation [28]. Further literature on SHACL following its standardisation can be roughly divided into two groups. The first group studies the formal properties and expressiveness of the non-recursive fragment [8]. Notable examples in this category are: the work by Delva et al. on data provenance [17], the work of Pareti et al. on satisfiability and (shape) containment [41], and the work of Leinberger et al. connecting the containment problem to description logics [34]. The second group of papers is concerned with proposing a suitable semantics for recursive SHACL [2, 6, 13, 14] or studying the complexity of certain problems for recursive SHACL under a chosen semantics [40]. First reports on practical applications and use-cases for SHACL include the study of expressivity of property constraints, as well as mining and extracting constraints in the context of large knowledge graphs such as Wikidata and DBpedia [19, 44]. Finally, the underlying ideas of SHACL where transposed to the setting of Property Graphs in a formalism called ProGS [47].

ShEx literature. ShEx was initially proposed in 2014 as a concise and human-readable language to describe, validate, and transform RDF data [43]. Its formal semantics was formally defined in [49]. The semantics of ShEx schemas combining recursion and negation was later presented in [9]. The current semantic specification of the ShEx language has been published as a W3C Community group report [42] and a new language version is currently being defined as part of the IEEE Working group on Shape Expressions[1]. As for practical applications, ShEx has been applied as a descriptive schema language through the Wikidata Schemas project[2]. Additional work went into extending ShEx to handle graph models that go beyond RDF, like WShEx to validate Wikibase graphs [29], ShEx-Star to handle RDF-Star and PShEx to handle property graphs [30]. While these works extend ShEx to (different types of) property graphs, they do not provide a common graph data model, nor compare schema languages, as we do.

[1]https://shex.io/shex-next/

[2]https://www.wikidata.org/wiki/Wikidata:WikiProject_Schemas

PG-Schema literature. PG-Schema, as introduced in [3], builds upon an earlier proposal of PG-Keys [4] to enhance schema support for property graphs, in the light of limited schema support in existing systems and the current version of the GQL standard [24]. It is currently being used in the GQL standardization process as a basis for a standard for property graph schemas.

Comparing RDF schema formalisms. In Chapter 7 of [32], the authors compare common features and differences between ShEx and SHACL and [31] presents a simplified language called S, which captures the essence of ShEx and SHACL. Tomaszuk [50] analyzes advances in RDF validation, highlighting key requirements for validation languages and comparing the strengths and weaknesses of various approaches.

Interoperability between schema graph formalisms. Interoperability between schema graph formalisms like RDF and Property Graphs remains challenging due to differences in structure and semantics. RDF focuses on triple-based modeling with formal semantics, while Property Graphs allow flexible annotation of relationships with properties. RDF-star [21] and RDF 1.2 [26] extend RDF 1.1 by enabling statements about triples, aligning more closely with labeled property graphs: for instance, RDF-star allows triples to function as subjects or objects, similar to how property graph edges carry properties.

By adopting *named graphs* [12], already RDF 1.1 provided a mechanism for making statements about (sub-)graphs. Likewise, different *reification* mechanisms have been proposed in the literature for RDF in order to "embed" meta-statements about triples (and graphs) in "vanilla" RDF graphs, ranging from the relatively verbose original W3C reification vocabulary as part of the original RDF specification, to more subtle approaches such as singleton property reification [36], which is close to the unique identifiers used for edges in most property graph models. Custom reification models are used, for instance, in Wikidata, to map Wikibase's property graph schema to RDF, cf. e.g. [19, 22]. There is also work on schema-independent and schema-dependent methods for transforming RDF into Property Graphs, providing formal foundations for preserving information and semantics [5]. All these approaches, in principle, facilitate general or specific mappings between RDF and property graphs, which is what the present paper tries to avoid by focusing on a common submodel.

There have been several prior proposals for uniying graph data models, rather then providing mappings between them. The One-Graph initiative [33] aims to bridge the different graph data models by promoting a unified graph data model for seamless interaction. Similarly, MilleniumDB's Domain Graph model [51] aims at covering RDF, RDF-star, and property graphs. These works seek a common *supermodel*, aiming to support both RDF and property graphs via more general solutions. In contrast, we aim at understanding the existing schema languages by studying them over a common submodel of RDF and property graphs.

Schemas for tree-structured data. The principle of defining (parts of) schemas as a set of pairs (sel, φ) is very prominent in schema languages for XML. A DTD [11] is essentially such a set of pairs in which *sel* selects nodes with a certain label, and φ describes the structure of their children. In XML Schema, the principle was used for defining key constraints (using *selectors* and *fields*) [20, Section 3.11.1]. The equally expressive language BonXai [35] is

based on writing the entire schema using such rules. Schematron [23] is another XML schema language that differs from grammar-based languages by defining patterns of assertions using XPath expressions [18]. It excels in specifying constraints across different branches of a document tree, where traditional schema paradigms often fall short. Schematron's rule-based structure, composed of phases, patterns, rules, and assertions, allows for the validation of documents.

RDF validation. Last, but not least, it should be noted that the requirement for (constraining) schema languages—besides ontology languages such as OWL and RDF Schema—in the Semantic Web community is much older than the more recent additions of SHACL and ShEx. Earlier proposals in a similar direction include efforts to add constraint readings of Description Logic axioms to OWL, such as OWL Flight [16] or OWL IC [48]. Another approach is Resource Shapes (ReSh) [45], a vocabulary for specifying RDF shapes. The authors of ReSh recognize that RDF terms originate from various vocabularies, and the ReSh shape defines the integrity constraints that RDF graphs are required to satisfy. Similarly, Description Set Profiles (DSP) [37] and SPARQL Inferencing Notation (SPIN) [27] are notable alternatives. While SHACL, ShEx, and ReSh share declarative, high-level descriptions of RDF graph content, DSP and SPIN offer additional mechanisms for validating and constraining RDF data, each with its own strengths and applications.

Implementations. Dozens of tools support graph data validation, including ShEx and SHACL. A comprehensive collaborative list of resources is available at: https://github.com/w3c-cg/awesome-semantic-shapes.

B Distilling the common data model

In this section we discuss the relationship between common graphs and the standard data models of the three schema formalisms—RDF and property graphs.

B.1 Comparison with RDF

Recall that an RDF graph is a set a triples in

$$(\mathsf{IRIs} \cup \mathsf{Blanks}) \times \mathsf{IRIs} \times (\mathsf{IRIs} \cup \mathsf{Blanks} \cup \mathsf{Literals})\,.$$

As explained in Section 2, common graphs can be naturally seen as finite sets of triples from

$$\mathcal{E} = (\mathcal{N} \times \mathcal{P} \times \mathcal{N}) \;\cup\; (\mathcal{N} \times \mathcal{K} \times \mathcal{V})\,,$$

with (E, ρ) corresponding to $E \;\cup\; \{(u, k, v) \mid \rho(u, k) = v\}$. In the RDF context, one would assume the following:

- $\mathcal{N} \subseteq \mathsf{IRIs} \cup \mathsf{Blanks}$,
- $\mathcal{P} \subseteq \mathsf{IRIs}$,
- $\mathcal{K} \subseteq \mathsf{IRIs}$,
- $\mathcal{V} = \mathsf{Literals}$.

However, the common graph data model does not refer to IRIs, Blanks, and Literals at all, because these are not part of the property graph data model. Unlike in RDF, a common graph may contain at most one tuple of the form (u, k, v) for each $u \in \mathcal{N}$ and $k \in \mathcal{K}$. This reflects the assumption that properties are single-valued, which is present in the property graph data model. The distinction between predicates and properties corresponds to the distinction between *datatype properties* and *object properties* in OWL [52].

In contrast to the RDF model, but in accordance with the perspective commonly taken in databases, both values and nodes are atomic. For nodes we completely abstract away from the actual representation of their identities. We do not even distinguish between IRIs and Blanks. An immediate consequence of this is that schemas do not have access to any information about the node other than the triples in which it participates. In particular, they cannot compare nodes with constants. This means that concrete nodes cannot be listed as validation targets; rather, validation targets must be specified in a generic way based on observable aspects of nodes, such as participation in some triples. This is a restriction with respect to the RDF data model, but it follows immediately from the same assumption made in the property graph data model. Importantly, the abstractions of ShEx and SHACL we provide can be easily extended to allow references to concrete nodes, if needed.

For values we take a more subtle approach: we assume a set $\mathcal{T}$ of value types, with each $\mathtt{v} \in \mathcal{T}$ representing a set $[\![\mathtt{v}]\!] \subseteq \mathcal{V}$. This captures uniformly data types, such as `integer` or `string`, and user-defined checks, such as interval bounds for numeric values or regular expressions for strings. Also, unlike for nodes, we allow comparisons of values with constants. On the other hand, the common graph data model does not include any binary relations over values, such as an order.

B.2 Comparison with property graphs

Let us recall the standard definition of property graphs [3].

Definition 14 (Property graph). *A* property graph *is a tuple* $(N, E, \pi, \lambda, \rho)$ *such that*

- N *is a finite set of nodes;*
- E *is a finite set of edges, disjoint from* N*;*
- $\pi : E \to (N \times N)$ *maps edges to their source and target;*
- $\lambda : (N \cup E) \to 2^{\mathcal{P}}$ *maps nodes and edges to finite sets of labels;*
- $\rho : (N \cup E) \times \mathcal{K} \rightharpoonup \mathcal{V}$ *is a finite-domain partial function mapping element-key pairs to values.*

A common graph $G = (E', \rho')$ can be easily represented as a property graph by letting

- $N = \mathsf{Nodes}(G)$,
- $E = E'$,
- $\pi = \{(e, (v_1, v_2)) \mid e = (v_1, p, v_2) \in E\}$,
- $\lambda = \{(e, \{p\}) \mid e = (v_1, p, v_2) \in E\} \cup \{(v, \emptyset) \mid v \in N\}$, and
- $\rho = \rho'$.

It is possible to characterise exactly the property graphs that are such representations of common graphs. These are the property graphs $(N, E, \pi, \lambda, \rho)$ for which it holds that:

(1) $\lambda(v) = \emptyset$ for all $v \in N$, and $\lambda(e)$ is a singleton for all $e \in E$,
(2) there cannot be two distinct edges $e_1, e_2 \in E$ such that $\pi(e_1) = \pi(e_2)$ and $\lambda(e_1) = \lambda(e_2)$, and
(3) $\rho(e, k)$ is undefined for all $e \in E$, $k \in \mathcal{K}$.

So, common graphs can be interpreted as restricted property graphs: no labels on nodes, single labels on edges, no parallel edges with the same label, and no properties on edges. All these restrictions are direct consequences of the nature of the RDF data model.

While these restrictions seem severe at a first glance, the resulting data model can actually easily simulate unrestricted property

graphs: labels on nodes can be simulated with the presence of corresponding keys, edges can be materialised as nodes if we need properties over edges or parallel edges with the same label. This means not only that common graphs can be used without loss of generality in expressiveness and complexity studies, but also that the corresponding restricted property graphs are flexible enough to be usable in practice, while additionally guaranteeing interoperability with the RDF data model.

B.3 Class information

The common graph data model does not have direct support for class information. The reason for this is that RDF and property graphs handle class information rather differently. In RDF, both class and instance information is part of the graph data itself: classes are elements of the graph, subclass-superclass relationships are represented as edges between classes, and membership relationships are represented as edges between elements and classes. In property graphs, the membership of a node in a class is indicated by a label put on the node. A node can belong to many classes, but the only way to say that class A is a subclass of class B is to ensure in the schema that each node with label A also has label B. That is,

- in property graphs, class membership information is available locally in a node, but consistency must be ensured by the schema,
- in RDF, obtaining class membership information requires navigating in the graph, but consistency is for free.

Clearly, both approaches have their merits, but when passing from one to the other data needs to be translated. This means that we cannot pick one of these approaches for the common data model while keeping it a natural submodel of both RDF and property graphs. Therefore, to reduce the complexity of this study, we do not include any dedicated features for supporting class information in our common data model. Note, however, that common graphs can support both these approaches indirectly: designated predicates can be used to represent membership and subclass relationships, and keys with a dummy value can simulate node labels.

C More on the core

In this section we prove Proposition 1. Recall that common shapes are defined by the grammar

$$\begin{aligned}
\varphi &::= \exists\, \pi \mid \exists^{\leq n}\, \pi_1 \mid \exists^{\geq n}\, \pi_1 \mid \exists\, \mathbb{c} \wedge \nexists\, \neg P \mid \varphi \wedge \varphi\ .\\
\mathbb{c} &::= \{\} \mid \{k : \mathbb{v}\} \mid \mathbb{c}\ \&\ \mathbb{c} \mid \mathbb{c} \mid \mathbb{c}\ .\\
\pi_0 &::= [k = c] \mid \neg[k = c] \mid \mathbb{c}\ \&\ \top \mid \neg(\mathbb{c}\ \&\ \top) \mid \pi_0 \cdot \pi_0\ .\\
\pi_1 &::= \pi_0 \cdot p \cdot \pi_0 \mid \pi_0 \cdot p^- \cdot \pi_0 \mid \pi_0 \cdot k \mid k^- \cdot \pi_0\ .\\
\bar{\pi} &::= \pi_0 \mid p \mid \bar{\pi}^- \mid \bar{\pi} \cdot \bar{\pi} \mid \bar{\pi} \cup \bar{\pi}\ .\\
\pi &::= \bar{\pi} \mid \bar{\pi} \cdot k \mid k^- \cdot \bar{\pi} \mid k^- \cdot \bar{\pi} \cdot k'\ .
\end{aligned}$$

where $n \in \mathbb{N}$, $P \subseteq_{fin} \mathcal{P}$, $k, k' \in \mathcal{K}$, $c \in \mathcal{V}$, and $p \in \mathcal{P}$. We will refer to PG-path expressions defined by the nonterminal π_0 in the grammar as *filters*.

The following two subsections describe the translations of common schemas to SHACL and ShEx. The translations are very similar but we include them both for the convenience of the reader.

C.1 Translation to SHACL

Lemma 1. *For each open content type $\mathbb{c}$ there is a SHACL shape $\varphi_{\mathbb{c}}$ such that $\mathcal{G}, v \vDash \varphi_{\mathbb{c}}$ iff $\rho(v) \in [\![\mathbb{c}]\!]$ for all $\mathcal{G} = (E, \rho)$ and $v \in \mathsf{Nodes}(\mathcal{G})$.*

Proof. For the content type $\top$ the corresponding SHACL shape is $\top$. For a content type of the form

$$\{k_1 : \mathbb{v}_1\}\ \&\ \{k_2 : \mathbb{v}_2\}\ \&\ \cdots\ \&\ \{k_m : \mathbb{v}_m\}\ \&\ \top\,,$$

the corresponding SHACL shape is

$$\exists\, k_1.\mathsf{test}(\mathbb{v}_1) \wedge \exists\, k_2.\mathsf{test}(\mathbb{v}_2) \wedge \cdots \wedge \exists\, k_m.\mathsf{test}(\mathbb{v}_m)\,.$$

Finally, every open content type different from $\top$ can be expressed as

$$(\mathbb{c}_1 \mid \cdots \mid \mathbb{c}_\ell)\ \&\ \top\,,$$

where each $\mathbb{c}_i$ is a content type of the form $\{k_1 : \mathbb{v}_1\}\ \&\ \{k_2 : \mathbb{v}_2\}\ \&\ \cdots\ \&\ \{k_m : \mathbb{v}_m\}$ for some m. The corresponding SHACL shape is

$$\varphi_1 \vee \cdots \vee \varphi_\ell\,,$$

where φ_i is the SHACL shape corresponding to the content type $\mathbb{c}_i\ \&\ \top$. □

Lemma 2. *For each filter π_0 there is a SHACL shape φ_{π_0} such that $\mathcal{G}, v \vDash \varphi_{\pi_0}$ iff $(v, v) \in [\![\pi_0]\!]^{\mathcal{G}}$ for all $\mathcal{G}$ and $v \in \mathsf{Nodes}(\mathcal{G})$.*

Proof. By Lemma 1, the claim holds for $\pi_0 = \mathbb{c}\ \&\ \top$. For $\{k : c\}$ the corresponding SHACL shape is $\exists k.\mathsf{test}(c)$. As SHACL shapes are closed under negation, the claim holds for $\neg\{k : c\}$ and $\neg(\mathbb{c}\ \&\ \top)$. Finally, concatenations of filters correspond to conjunctions of shapes, so the claim follows because SHACL shapes are closed under conjunction. □

Lemma 3. *For each common shape of the form $\exists\, \pi$ there is a SHACL shape $\varphi_{\exists \pi}$ such that $\mathcal{G}, v \vDash \varphi_{\exists \pi}$ iff $\mathcal{G}, v \vDash \exists\, \pi$ for all $\mathcal{G}$ and $v \in \mathsf{Nodes}(\mathcal{G}) \cup \mathsf{Values}(\mathcal{G})$.*

Proof. Let us first look at common shapes of the form $\exists\, \pi$ where π is a concatenation of filters and atomic path expressions of the form p, p^-, k, or k^-. Without loss of generality we can assume that the concatenation ends with a filter or with k. We proceed by induction on the length of the concatenation. The base cases are $\exists \pi_0$ and $\exists k$, which correspond to φ_{π_0} (Lemma 2) and $\exists k.\top$. For $\exists\, \pi_0 \cdot \pi$ we can take $\varphi_{\pi_0} \wedge \varphi_{\exists \pi}$. For $\exists\, p \cdot \pi$ we can take $\exists p.\varphi_{\exists \pi}$, and similarly for $\exists\, p^- \cdot \pi$ and $\exists\, k^- \cdot \pi$.

The general case follows because SHACL shapes are closed under union. Indeed, because our PG-path expressions are star-free, we can assume without loss of generality that in each common shape of the form $\exists\, \pi$, the PG-path expression $\bar{\pi}$ underlying π is a union of concatenations of filters and atomic path expressions of the form p or p^-. Then, for

$$\exists\, k^- \cdot (\pi^1 \cup \cdots \cup \pi^m) \cdot k'$$

we can take

$$\varphi_{\exists k^- \cdot \pi^1 \cdot k'} \vee \cdots \vee \varphi_{\exists k^- \cdot \pi^m \cdot k}\,.$$

Simiarly for $\exists\, k^- \cdot (\pi^1 \cup \cdots \cup \pi^m)$, $\exists\, (\pi^1 \cup \cdots \cup \pi^m) \cdot k'$, and $\exists\, (\pi^1 \cup \cdots \cup \pi^m)$. □

Lemma 4. *For each common shape φ there is a SHACL shape $\hat{\varphi}$ such that $\mathcal{G}, v \vDash \varphi$ iff $\mathcal{G}, v \vDash \hat{\varphi}$ for all $\mathcal{G}$ and $v \in \mathsf{Nodes}(\mathcal{G}) \cup \mathsf{Values}(\mathcal{G})$.*

Proof. Because SHACL shapes are closed under conjunction, it suffices to prove the claim for the atomic common shapes of the forms $\exists\,\pi$, $\exists^{\leq n}\pi_1$, $\exists^{\geq n}\pi_1$, and $\exists\mathbb{c} \wedge \nexists\neg P$. The first case follows from Lemma 3.

Let us look at common shapes of the form $\exists^{\geq n}\pi_1$. If $n = 0$ we can simply take $\top$. Suppose $n > 0$. Then, for

$$\exists^{\geq n}\pi_0 \cdot p \cdot \pi_0'$$

we can take

$$\varphi_{\pi_0} \wedge \exists^{\geq n} p.\varphi_{\pi_0'},$$

and similarly for $\exists^{\geq n}\pi_0 \cdot p^- \cdot \pi_0'$, $\exists^{\geq n}\pi_0 \cdot k^- \cdot \pi_0'$, and $\exists^{\geq n}\pi_0 \cdot k$ (using $\top$ instead of $\varphi_{\pi_0'}$).

Next, we consider common shapes of the form $\exists^{\leq n}\pi_1$. For

$$\exists^{\leq n}\pi_0 \cdot p \cdot \pi_0'$$

we can take

$$\neg\varphi_{\pi_0} \vee \exists^{\leq n} p.\varphi_{\pi_0'},$$

and similarly for $\exists^{\leq n}\pi_0 \cdot p^- \cdot \pi_0'$, $\exists^{\leq n}\pi_0 \cdot k^- \cdot \pi_0'$, and $\exists^{\leq n}\pi_0 \cdot k$ (again, using $\top$ instead of $\varphi_{\pi_0'}$).

Finally, let us consider a common shape of the form $\exists\,\mathbb{c} \wedge \nexists\neg P$. Suppose first that

$$\mathbb{c} = \{\}\,.$$

Then, the corresponding SHACL shape is simply

$$\mathsf{closed}(P)\,.$$

Next, suppose that

$$\mathbb{c} = \{k_1 : \mathbb{v}_1\}\,\&\,\ldots\,\&\,\{k_m : \mathbb{v}_m\}\,.$$

Then, the corresponding SHACL shape is

$$\exists k_1.\mathsf{test}(\mathbb{v}_1) \wedge \cdots \wedge \exists k_m.\mathsf{test}(\mathbb{v}_m) \wedge \mathsf{closed}\big(\{k_1,\ldots,k_m\} \cup P\big)\,.$$

In general, as in Lemma 1, we can assume that

$$\mathbb{c} = \mathbb{c}_1 \mid \ldots \mid \mathbb{c}_m$$

where each $\mathbb{c}_i$ is of one of the two forms considered above. The corresponding SHACL shape is then

$$\varphi_1 \vee \cdots \vee \varphi_m$$

where φ_i is the SHACL shape corresponding to $\exists\mathbb{c}_i \wedge \nexists\neg P$, obtained as described above. □

Lemma 5. *For every common schema there is an equivalent SHACL schema.*

Proof. Let $\mathcal{S}$ be a common schema. We obtain an equivalent SHACL schema $\mathcal{S}'$ by translating each $(sel, \varphi) \in \mathcal{S}$ to (sel', φ') such that for all $\mathcal{G}$ and $v \in \mathcal{N} \cup \mathcal{V}$,

$$\mathcal{G}, v \vDash sel \text{ implies } \mathcal{G}, v \vDash \varphi$$
$$\text{iff}$$
$$\mathcal{G}, v \vDash sel' \text{ implies } \mathcal{G}, v \vDash \varphi'.$$

Recall that *sel* is a common shape of one of the following forms:

$$\exists\,k,\ \exists\,p\cdot\pi,\ \exists\,p^-\cdot\pi,\ \exists\,\{k:c\}\cdot\pi,\ \exists\,\big(\{k:\mathbb{v}\}\,\&\,\top\big)\cdot\pi,\ \exists\,k^-\cdot\pi\,.$$

For sel' we take, respectively,

$$\exists\,k.\top,\quad \exists\,p.\top,\quad \exists\,p^-.\top,\quad \exists\,k.\top,\quad \exists\,k.\top,\quad \exists\,k^-.\top\,.$$

For φ' we take $\neg\varphi_{sel} \vee \hat{\varphi}$ where φ_{sel} is obtained by applying Lemma 3 to *sel*, and $\hat{\varphi}$ is obtained by applying Lemma 4 to φ. □

C.2 Translation to ShEx

Lemma 6. *For each open content type $\mathbb{c}$ there is a ShEx shape $\varphi_{\mathbb{c}}$ such that $\mathcal{G}, v \vDash \varphi_{\mathbb{c}}$ iff $\rho(v) \in [\![\mathbb{c}]\!]$ for all $\mathcal{G} = (E, \rho)$ and $v \in \mathsf{Nodes}(\mathcal{G})$.*

Proof. For the content type $\top$ the corresponding ShEx shape is $\{\top\}$.

For a content type of the form

$$\{k_1 : \mathbb{v}_1\}\,\&\,\cdots\,\&\,\{k_m : \mathbb{v}_m\}\,\&\,\top\,,$$

the corresponding ShEx shape is

$$\{k_1.\mathsf{test}(\mathbb{v}_1)\,;\top\} \wedge \cdots \wedge \{k_m.\mathsf{test}(\mathbb{v}_m)\,;\top\}\ .$$

Finally, every other open content type can be expressed as

$$(\mathbb{c}_1 \mid \cdots \mid \mathbb{c}_\ell)\,\&\,\top\,,$$

where each $\mathbb{c}_i$ has the form $\{k_1 : \mathbb{v}_1\}\,\&\,\cdots\,\&\,\{k_m : \mathbb{v}_m\}$ for some m. The corresponding ShEx shape is

$$\varphi_1 \vee \cdots \vee \varphi_\ell\,,$$

where φ_i is the ShEx shape corresponding to the content type $\mathbb{c}_i\,\&\,\top$. □

Lemma 7. *For each filter π_0 there is a ShEx shape φ_{π_0} such that $\mathcal{G}, v \vDash \varphi_{\pi_0}$ iff $(v, v) \in [\![\pi_0]\!]^{\mathcal{G}}$ for all $\mathcal{G}$ and $v \in \mathsf{Nodes}(\mathcal{G})$.*

Proof. By Lemma 6, the claim holds for $\pi_0 = \mathbb{c}\,\&\,\top$. For $\{k : c\}$ the corresponding ShEx shape is $\{k.\mathsf{test}(c); \top\}$. Because ShEx shapes are closed under negation, the claim also holds for $\neg\{k : c\}$ and $\neg(\mathbb{c}\,\&\,\top)$. Finally, concatenations of filters correspond to conjunctions of shapes, so the claim follows because ShEx shapes are closed under conjunction. □

Lemma 8. *For each common shape of the form $\exists\,\pi$ there is a ShEx shape $\varphi_{\exists\pi}$ such that $\mathcal{G}, v \vDash \varphi_{\exists\pi}$ iff $\mathcal{G}, v \vDash \exists\,\pi$ for all $\mathcal{G}$ and $v \in \mathsf{Nodes}(\mathcal{G}) \cup \mathsf{Values}(\mathcal{G})$.*

Proof. Let us first look at common shapes of the form $\exists\,\pi$ where π is a concatenation of filters and atomic path expressions of the form p, p^-, k, or k^-. Without loss of generality we can assume that the concatenation ends with a filter or with k. We proceed by induction on the length of the concatenation. The base cases are $\exists\pi_0$ and $\exists k$, which correspond to φ_{π_0} (Lemma 7) and $\{k.\{\top\}\,;\top\}$, respectively. For $\exists\,\pi_0\cdot\pi$ we can take $\varphi_{\pi_0} \wedge \varphi_{\exists\pi}$. For $\exists\,p\cdot\pi$ we can take $\{p.\varphi_{\exists\pi}\,;\top\}$, and similarly for $\exists\,p^-\cdot\pi$ and $\exists\,k^-\cdot\pi$.

The general case follows because ShEx shapes are closed under union. Indeed, because our PG-path expressions are star-free, we can assume without loss of generality that in each common shape of the form $\exists\,\pi$, the PG-path expression $\bar{\pi}$ underlying π is a union of concatenations of filters and atomic path expressions of the form p or p^-. Then, for

$$\exists\,k^-\cdot(\pi^1 \cup \cdots \cup \pi^m)\cdot k'$$

we can take

$$\varphi_{\exists k^-\cdot\pi^1\cdot k'} \vee \cdots \vee \varphi_{\exists k^-\cdot\pi^m\cdot k}\,.$$

Simiarly for $\exists\,k^-\cdot(\pi^1 \cup \cdots \cup \pi^m)$, $\exists\,(\pi^1 \cup \cdots \cup \pi^m)\cdot k'$, and $\exists\,(\pi^1 \cup \cdots \cup \pi^m)$. □

Lemma 9. *For each common shape φ there is a ShEx shape $\hat{\varphi}$ such that $\mathcal{G}, v \vDash \varphi$ iff $\mathcal{G}, v \vDash \hat{\varphi}$ for all $\mathcal{G}$ and $v \in \mathsf{Nodes}(\mathcal{G}) \cup \mathsf{Values}(\mathcal{G})$.*

Proof. Because ShEx shapes are closed under conjunction, it suffices to prove the claim for the atomic common shapes of the forms $\exists\,\pi$, $\exists^{\leq n}\pi_1$, $\exists^{\geq n}\pi_1$, and $\exists\mathbb{c} \wedge \nexists\neg P$. The first case follows from Lemma 8.

Let us look at common shapes of the form $\exists^{\geq n}\pi_1$. If $n = 0$ we can simply take $\{\top\}$. Suppose $n > 0$. Then, for

$$\exists^{\geq n}\pi_0 \cdot p \cdot \pi_0'$$

we can take

$$\varphi_{\pi_0} \wedge \left\{\left(p.\varphi_{\pi_0'}\right)^n ; \top\right\},$$

and similarly for $\exists^{\geq n}\pi_0 \cdot p^- \cdot \pi_0'$, $\exists^{\geq n}\pi_0 \cdot k^- \cdot \pi_0'$, and $\exists^{\geq n}\pi_0 \cdot k$ (using $\{\top\}$ instead of $\varphi_{\pi_0'}$).

Next, we consider common shapes of the form $\exists^{\leq n}\pi_1$. For

$$\exists^{\leq n}\pi_0 \cdot p \cdot \pi_0'$$

we can take

$$\neg\varphi_{\pi_0} \vee \neg\left\{\left(p.\varphi_{\pi_0'}\right)^{n+1} ; \top\right\}$$

and similarly for $\exists^{\leq n}\pi_0 \cdot p^- \cdot \pi_0'$, $\exists^{\leq n}\pi_0 \cdot k^- \cdot \pi_0'$, and $\exists^{\leq n}\pi_0 \cdot k$ (again, using $\{\top\}$ instead of $\varphi_{\pi_0'}$).

Before we move on, let us introduce a bit of syntactic sugar. For a set $Q = \{q_1, q_2, \ldots, q_n\} \subseteq \mathcal{P} \cup \mathcal{K}$ we write Q^* for the triple expression $\left(q_1.\{\top\} \mid q_2.\{\top\} \mid \ldots \mid q_n.\{\top\}\right)^*$.

We are now ready to consider a common shape of the form $\exists\,\mathbb{c} \wedge \nexists\neg P$. Suppose first that

$$\mathbb{c} = \{\}\,.$$

Then, the corresponding ShEx shape is simply

$$\left\{P^* ; \left(\neg\emptyset^-\right)^*\right\}\ .$$

Next, suppose that

$$\mathbb{c} = \{k_1 : \mathbb{v}_1\}\ \&\ \ldots\ \&\ \{k_m : \mathbb{v}_m\}\,.$$

Then, the corresponding ShEx shape is

$$\varphi_{\mathbb{c}\&\top} \wedge \left\{\{k_1, \ldots, k_m\}^* ; P^* ; \left(\neg\emptyset^-\right)^*\right\}\ ,$$

where $\varphi_{\mathbb{c}\&\top}$ is obtained from Lemma 6. In general, as in Lemma 6, we can assume that

$$\mathbb{c} = \mathbb{c}_1 \mid \ldots \mid \mathbb{c}_m$$

where each $\mathbb{c}_i$ is of one of the two forms considered above. The corresponding ShEx shape is then

$$\varphi_1 \vee \cdots \vee \varphi_m$$

where φ_i is the ShEx shape corresponding to $\exists\mathbb{c}_i \wedge \nexists\neg P$, obtained as described above. □

Lemma 10. *For every common schema there is an equivalent ShEx schema.*

Proof. Let $\mathcal{S}$ be a common schema. We obtain an equivalent ShEx schema $\mathcal{S}'$ by translating each $(sel, \varphi) \in \mathcal{S}$ to (sel', φ') such that for all $\mathcal{G}$ and $v \in \mathcal{N} \cup \mathcal{V}$,

$$\begin{gathered}\mathcal{G}, v \vDash sel \text{ implies } \mathcal{G}, v \vDash \varphi \\ \text{iff} \\ \mathcal{G}, v \vDash sel' \text{ implies } \mathcal{G}, v \vDash \varphi'.\end{gathered}$$

Recall that *sel* is a common shape of one of the following forms:

$$\exists\,k\,,\ \exists\,p \cdot \pi\,,\ \exists\,p^- \cdot \pi\,,\ \exists\,\{k : c\} \cdot \pi\,,\ \exists\left(\{k : \mathbb{v}\}\ \&\ \top\right) \cdot \pi\,,\ \exists\,k^- \cdot \pi\,.$$

If *sel* is of the form

$$\exists\,k\,, \quad \exists\,\{k : c\} \cdot \pi\,, \quad \text{or} \quad \exists\left(\{k : \mathbb{v}\}\ \&\ \top\right) \cdot \pi\,,$$

for sel' we take $\{k.\{\top\} ; \top\}$. In the remaining cases, we take, respectively,

$$\{p.\{\top\} ; \top\}\,, \quad \{p^-.\{\top\} ; \top\}\,, \quad \{k^-.\{\top\} ; \top\}\,.$$

For φ' we take $\neg\varphi_{sel} \vee \hat{\varphi}$ where φ_{sel} is obtained by applying Lemma 8 to *sel*, and $\hat{\varphi}$ is obtained by applying Lemma 9 to φ. □