ProGS: Property Graph Shapes Language

Philipp Seifer[1](✉), Ralf Lämmel[1], and Steffen Staab[2,3]

[1] The Software Languages Team, University of Koblenz-Landau, Koblenz, Germany
{pseifer,laemmel}@uni-koblenz.de

[2] Institute for Parallel and Distributed Systems, University of Stuttgart, Stuttgart, Germany
steffen.staab@ipvs.uni-stuttgart.de

[3] Web and Internet Science Research Group, University of Southampton, Southampton, England

Abstract. Knowledge graphs such as Wikidata are created by a diversity of contributors and a range of sources leaving them prone to two types of errors. The first type of error, falsity of facts, is addressed by property graphs through the representation of provenance and validity, making triples occur as first-order objects in subject position of metadata triples. The second type of error, violation of domain constraints, has not been addressed with regard to property graphs so far. In RDF representations, this error can be addressed by shape languages such as SHACL or ShEx, which allow for checking whether graphs are valid with respect to a set of domain constraints. Borrowing ideas from the syntax and semantics definitions of SHACL, we design a shape language for property graphs, ProGS, which allows for formulating shape constraints on property graphs including their specific constructs, such as edges with identities and key-value annotations to both nodes and edges. We define a formal semantics of ProGS, investigate the resulting complexity of validating property graphs against sets of ProGS shapes, compare with corresponding results for SHACL, and implement a prototypical validator that utilizes answer set programming.

Keywords: Property graphs · Graph validation · SHACL

1 Introduction

Knowledge graphs such as Wikidata [20] require a data model that allows for the representation of data annotations. While property graphs serve well as data models for representing such knowledge graphs, they lack sufficient means for validation against domain constraints, for instance required provenance annotations. The shapes constraint language SHACL [22] was introduced to allow for validating knowledge graphs that use the RDF data model [21]. Wikidata and other knowledge graphs, however, make use of triples in subject position to represent provenance metadata, such as references or dates, going beyond the capabilities of the RDF framework. Similar to extensions of RDF, such as RDF* [10] or aRDF [19], property graphs are a promising data model for meeting the modelling needs of annotated knowledge graphs. Recent property-graph data

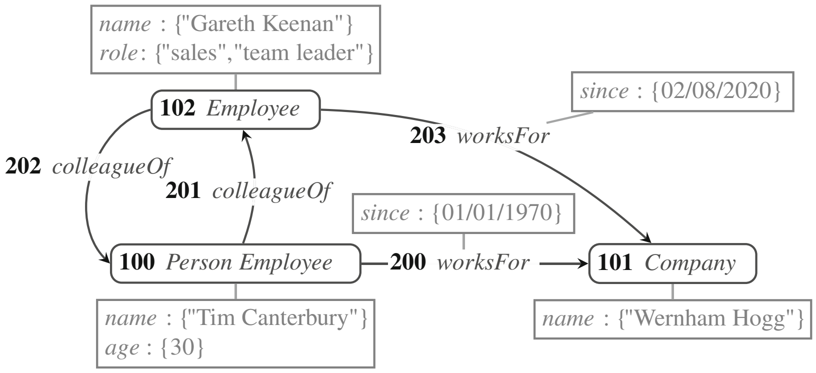

Fig. 1. Example property graph G_{office} showing employment relationships in G-CORE style: Nodes are depicted as rounded boxes. Each node has exactly one identifier, e.g., 100 or 101, and it has zero or more labels, e.g., $\{Person, Employee\}$ or $\{Company\}$. Each edge has an identifier, e.g., 200, as well as zero or more labels, e.g., $\{worksFor\}$. Both nodes and edges may have a set of affiliated properties (key-value pairs shown in rectangular boxes), e.g., $\{age : \{30\}\}$ or $\{since : \{01/01/1970\}\}$.

model (and query language) proposals include G-CORE [2] and the upcoming GQL standard [12], as well as the recently established openCypher standard [16]. They have attracted a lot of research interest and popularity in practical use-cases [18].

Property-graph models differ from RDF in substantial ways, featuring edges with identities (allowing multiple edges between nodes with the same labels) and property annotations (that is key-value annotations) on edges. A schema or shape-based validation language must account for these differences. While there exist efforts to formally define property graph schema languages [3,11], and some practical implementations support simple schemata [14] (e.g., uniqueness constraints) or even enable SHACL validation for RDF compatible subsets of the data graph [15], they do not allow for expressing shape constraints involving all elements of property graphs. In particular, existing approaches lack support for qualified number restrictions over edge identities, path expressions or the targeted validation of edges.

Consider the example graph G_{office} depicting employment relationships in Fig. 1. Some of the nodes and edges have property annotations. The edge with identity 200, for example, has the annotation *since* with values $\{01/01/1970\}$. One may wish to define *shapes* to require that all edges labelled *worksFor* have such metadata annotations. Shapes that constrain *Employee* or *Company* and their interrelationships will lead to recursive descriptions and thus require a corresponding semantics. Like [6], we adopt a model-based formal semantics based on the notion of (partial) assignments that map nodes and edges to sets of shape names and constitute the basis for a three-valued evaluation function.

Contributions. We present ProGS, a shape language for property graphs that allows for formulating domain constraints and that significantly extends SHACL to property graph data models. ProGS comprises property-graph specific features, including shapes for edges with identities, qualified number restrictions over such edges and constraints on properties and their values. We define the formal semantics for validating graphs with ProGS shapes, including cyclic, recursive shape references, based on the notion of partial faithful assignments inspired by [6]. We analyse the complexity of validating property graphs against sets of ProGS constraints. We show that ProGS validation is NP complete, thus remaining in the same complexity class as SHACL while increasing expressiveness. We provide a prototypical reference implementation relying on answer set programming, available on GitHub.

Outline. The remainder of this paper is structured as follows. Section 2 gives a short overview of property graph data models. In Sect. 3 we define the abstract syntax and semantics of ProGS, including assignment-based validation of graphs against a set of ProGS shapes. Section 4 analyses the complexity of the ProGS graph validation problem. Section 5 investigates implementation approaches for ProGS and introduces a prototypical implementation relying on an encoding of the validation problem as an answer set program. Section 6 discusses related work and Sect. 7 concludes the paper.

2 Foundations

Before providing a working definition of property graphs as the basis of ProGS, we compare existing property graph models to determine essential features. To this end, consider Table 1. We compare the property graph models underlying the graph query languages G-CORE [2], Cypher [9], Gremlin [4], and PGQL [17]; we also include the RDF [21] data model and RDF* [10] as a point of reference.

We use the example depicted in Fig. 2, an excerpt from Wikidata, to illustrate the differences between property graphs, RDF and RDF*. The defining feature of property graphs are properties, that are key-value pairs, on edges and nodes. For example, *point in time* in Fig. 2 could be represented as such a property annotation for the edge labelled *nominated for*. Property keys are strings, while value domains vary between approaches, ranging from simple scalar values and strings to lists or maps of values. The key differences to RDF arise from the fact that edges in property graphs have identities. The edge *nominated for*, for example, would have a unique identity acting as a target for property annotations. While this is not possible in plain RDF, node properties can be simulated through edges to literal nodes. RDF* extends RDF by introducing triples that are first-order (first-order) objects, meaning they can occur in both subject and object position of other triples. This importantly subsumes edge properties, again through an encoding of literal nodes. While not using RDF*, Wikidata also allows for annotations on edges referencing other resources. This highlights the key difference between support for triples in subject position of arbitrary triples and property

Table 1. Comparison of feature support for common property graph models and RDF. (a) using triples with literals as objects (b) triples in subject position of triples with literals as objects.

	G-CORE	Cypher	Gremlin	PGQL	RDF*	RDF
Nodes as first-order objects	+	+	+	+	+	+
Node properties	+	+	+	+	+[a]	+[a]
Node labels	set	set	none	set	set (rdf:type)	set (rdf:type)
Edges as first-order objects	+	+	+	+	+	-
Triples as first-order objects	−	−	−	−	+	−
Edge properties	+	+	+	+	+[b]	−
Edge labels	set	single	single	set	single	single
Paths as first-order objects	+	−	−	−	−	−
Path properties	+	−	−	−	−	−
Path labels	set	−	−	−	−	−

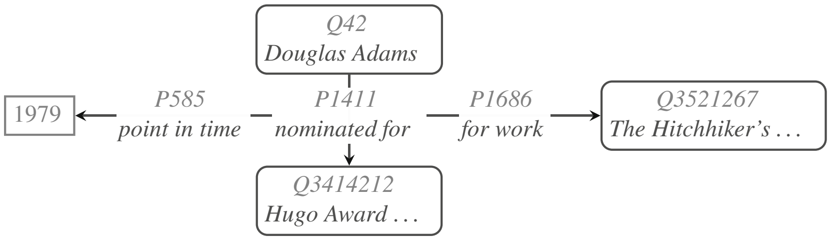

Fig. 2. Excerpt from Wikidata.

annotations: While *point in time* could be represented as a property annotation on the *nominated for* edge, *for work* could not.

There are some further differences between the various property graph models. Support for labels differs between approaches ranging from sets of labels on both nodes and edges (G-CORE, PGQL) to no support for node labels in Gremlin and single edge types in both Gremlin and Cypher. Finally, only G-CORE features paths as first-order objects, i.e., paths that can be annotated with property annotations and labels.

2.1 Definition of Property Graphs

The formalization of the property graph model we use as a basis for the definition of ProGS is based on the data model presented for G-CORE [2]. We do not consider first-class paths, and instead restrict the model to the core subset shared with other property graph models as discussed in the previous section. In terms of value domains in properties, we provide exemplary support for the types `string`, `int` and `date`, without loss of generality.

Let the set of labels $L = L_N \cup L_E$ where L_N is an infinite set of node labels and L_E an infinite set of edge labels. As a matter of convention, we use

CamelCase for all $l_N \in L_N$ and *camelCase* for all $l_E \in L_E$. Let K be an infinite set of property names (or keys) and V an infinite set of literal values from the union of sets in $T \in \{\texttt{int}, \texttt{string}, \texttt{date}\}$. We refer to elements of T as the type of the respective value. Let furthermore $\mathrm{FSET}(X)$ denote all finite subsets of a set X.

Definition 1 (Property Graph). *A property graph is a tuple* $G = (N, E, \rho, \lambda, \sigma)$*, where* N *denotes a set of node identifiers and* E *a set of edge identifiers, with* $N \cap E = \emptyset$*,* $\rho : E \to (N \times N)$ *is a total function,* $\lambda : (N \cup E) \to \mathit{FSET}(L)$ *is a total function,* $\sigma : (N \cup E) \times K \to \mathit{FSET}(V)$ *is a total function for which a finite set of tuples* $(x, k) \in (N \cup E) \times K$ *exists such that* $\sigma(x, k) \neq \emptyset$*.*

A property graph consists of a set of nodes $n \in N$ and edges $e \in E$, where ρ maps elements of E to pairs of nodes. The function λ maps nodes and edges to all assigned labels $l \in L$ and likewise the function σ maps pairs of nodes or edges, and property names to the property values assigned to them. The example in Fig. 3 shows the property graph visualized in Fig. 1 using the formal definition. Note that we omit infinitely many elements of the domain of σ that are mapped to $\emptyset$.

$$\begin{aligned}
N &= \{100, 101, 102\}\\
E &= \{200, 201, 202, 203\}\\
\rho &= \{200 \mapsto (100, 101), 201 \mapsto (100, 102), 202 \mapsto (102, 100), 203 \mapsto (102, 101)\}\\
\lambda &= \{100 \mapsto \{\mathit{Person}, \mathit{Employee}\}, 101 \mapsto \{\mathit{Company}\}, 102 \mapsto \{\mathit{Employee}\},\\
&\qquad 200 \mapsto \{\mathit{worksFor}\}, 201 \mapsto \{\mathit{colleagueOf}\}, 202 \mapsto \{\mathit{colleagueOf}\}\\
&\qquad 203 \mapsto \{\mathit{worksFor}\}\}\\
\sigma &= \{(100, \mathit{name}) \mapsto \{\text{"Tim Canterbury"}\}, (100, \mathit{age}) \mapsto \{30\},\\
&\qquad (101, \mathit{name}) \mapsto \{\text{"Wernham Hogg"}\}, (102, \mathit{name}) \mapsto \{\text{"Gareth Keenan"}\},\\
&\qquad (102, \mathit{role}) \mapsto \{\text{"sales"}, \text{"team leader"}\}, (200, \mathit{since}) \mapsto \{01/01/1970\},\\
&\qquad (203, \mathit{since}) \mapsto \{02/08/2020\}\}
\end{aligned}$$

Fig. 3. Formal model for the example property graph G_{office} rendered in Fig. 1.

3 Shapes for Property Graphs

Our shape language for property graph validation, called ProGS, has been inspired by SHACL [22], the W3C recommendation for writing and evaluating RDF graph validation constraints. More specifically, we base the ProGS shape language on the abstract syntax proposed by [6], which formalizes a syntactic core of SHACL. Corman et al. [6] also defined a formal semantics for this syntactic core that addresses recursion, in particular. We facilitate the understanding of differences between SHACL and ProGS by colour coding. Expressions that we borrow from SHACL will be displayed in black font, while novel expressions will be coded in blue font.

3.1 Requirements on a Property Graph Shapes Language

Requirements for our target language stem from the differences between the RDF and property graph data models, which we mentioned in Sect. 2. Table 2 explains how RDF may be mapped to the G-CORE property graph model. Based on this mapping we design ProGS to adopt language constructs from SHACL. The reader may note that this mapping includes some design decisions that are not unique, e.g., we interpret class instantiations as corresponding to G-CORE labellings of nodes. We follow a simplification of the third mapping $\mathcal{IM}_3$ discussed in [3], e.g., by excluding blank nodes.

Table 2. Sketching correspondences between the RDF and G-CORE graph models.

Description	RDF	G-CORE/ProGS
Node id i	IRI i	$i \in N$
Node n has label l	n rdf:type l.	$l \in \lambda(n)$
Node n has key k with value v	$n\,k\,v$.	$v \in \sigma(n, k)$
Edge id i	not available	$i \in E$
Edge label l, in triple $s\,p\,o$.	$s\,l\,o$	$l \in \lambda(p)$
Edge e has key k with value v	not available	$v \in \sigma(e, k)$
Triple $s\,p\,o$.	$s\,p\,o$.	$p \in \lambda(i), \rho(i) = (s, o)$

Edges in property graphs have identities necessitating two distinct kinds of shapes for nodes (R1) and for edges (R2) as well as two kinds of qualified number restrictions for nodes, counting edges (R3) and counting reachable nodes via some path (R4). Property annotations require dedicated constraints dealing with the set of values reachable via a specific key, for both nodes (R5) and edges (R6). The presence of properties must also be considered for constraints that include comparison operations (R7). Lastly, the existence of certain properties, or properties with certain values, also require new means of targeting nodes and edges in target queries (R8).

3.2 Definition of Shapes

Intuitively, a shape defines constraints on how certain nodes or edges in a graph are formed. As both nodes and edges in property graphs have identities, we define *node shapes* that apply to nodes and *edge shapes* that apply to edges. Each shape is a triple consisting of a shape name, a constraint, and a target query defining which nodes or which edges of a graph must conform to the shape, i.e., fulfil all of its constraints, for the graph to be considered in conformance with the shape.

Example 1. The node shape ${}_N\langle \mathit{EmployeeShape}, \mathit{Person}, \mathit{Employee}\rangle$ is a triple with the shape name *EmployeeShape*, the constraint *Person*, which requires that

$$\begin{aligned}
[\![\bot]\!]_G &= \emptyset \\
[\![n]\!]_G &= \{n\} \\
[\![l_N]\!]_G &= \{n \mid n \in N \land l_N \in \lambda(n)\} \\
[\![k]\!]_G &= \{n \mid n \in N \land \sigma(n,k) \neq \emptyset\} \\
[\![k : v]\!]_G &= \{n \mid n \in N \land v \in \sigma(n,k)\}
\end{aligned}$$

Fig. 4. Evaluation of target node queries.

each graph node assigned this shape has the label *Person*, and the target query *Employee*, meaning all nodes with the label *Employee* are targets of this shape. For the graph G_{office} in Fig. 1, node 100 conforms to this shape, whereas node 102 does not, lacking the *Employee* label. Given that at least one target node does not conform to the constraint, the entire graph does not conform to *EmployeeShape*.

As shown in the first example, we use ${}_N\langle s_N, \phi_N, q_N\rangle$ to indicate triples that are node shapes and use ${}_E\langle s_E, \phi_E, q_E\rangle$ to refer to triples that are edge shapes.

Before defining the components of shapes, we define the syntax of path expressions p in Eq. (1) in analogy to property path expressions defined in SHACL [22], which are in turn based on path expressions in the SPARQL query language. A path expression, when evaluated on a starting node, describes the set of nodes reachable from this node via paths that match the path expression.

$$p ::= \ l_E \mid p^- \mid p_1/p_2 \mid p_1 || p_2 \mid p* \mid p+ \mid ?p \tag{1}$$

Path expressions may include edge labels l_E, inverse paths p^-, path sequences p_1/p_2, alternate paths $p_1||p_2$ and zero or more ($p*$), one or more ($p+$) and zero or one ($?p$) expressions. Note the minor difference to paths in RDF graphs, in that edges in property graphs may have multiple labels.

Example 2. The path *worksFor*/*worksFor*$^-$ describes the set of all colleagues of a starting node n (including n itself), by first finding all employers of n (i.e., nodes reachable from n via an edge with label *worksFor*) and then all employees of those employers (i.e., nodes with incoming *worksFor* edges). For the graph G_{office} in Fig. 1 and starting node 100, the result of evaluating this path would be the same as evaluating *colleagueOf*$*$, namely the set $\{100, 102\}$.

Let the set of shapes $S = S_N \cup S_E$ consist of node and edge shapes and the set of shape names be called Names(S). A node shape is a tuple ${}_N\langle s_N, \phi_N, q_N\rangle$ consisting of a shape name $s_N \in \text{Names}(S_N)$, a node constraint ϕ_N and a query for target nodes q_N. A query for target nodes is either $\bot$, meaning the query has no targets, an explicitly targeted node $n \in N$, all nodes with label $l_N \in L_N$, all nodes with property $k \in K$ or possibly further constrained as $k : v$ by a concrete value $v \in V$. The syntax of target node queries q_N is summarized in Eq. (2). We write $[\![q_N]\!]_G$ for the evaluation of a target node query, which is defined in Fig. 4.

$$q_N ::= \ \bot \mid n \mid l_N \mid k \mid k : v \tag{2}$$

Example 3. The target query $q_N = Employee$ targets all nodes that are labelled with the label *Employee*. The set of targets when evaluating q_N on the example graph G_{office} in Fig. 1 is therefore $[\![q_N]\!]_{G_{\text{office}}} = \{100, 102\}$.

Node constraints ϕ_N essentially specify which outgoing or incoming edges, which labels, or which properties a targeted node must have. Assuming $s_N \in S_N$, $n \in N$, $l_N \in L_N$, $k \in K$, $i \in \mathbb{N}$, comparison operations $\odot$ for sets or singleton sets (e.g., $=$, $<$, $\subset$) and arbitrary value predicate functions $f : V \rightarrow \{0,1\}$ such as ≥ 0, $\neq 19$, or type restrictions for a specific data type such as `int`, `string` or `date`, the syntax of node constraints ϕ_N is defined as in Eq. (3).

$$\begin{aligned} \phi_N ::= \ & \top \mid s_N \mid n \mid l_N \mid \neg\phi_N \mid \phi_N^1 \wedge \phi_N^2 \mid \geqslant_i p.\phi_N \mid \odot(p_1, p_2) \\ & \mid \geqslant_i k.f \mid \geqslant_i^{\leftarrow} \phi_E \mid \geqslant_i^{\rightarrow} \phi_E \mid \odot(p_1, k_1, p_2, k_2) \mid \odot(k_1, k_2) \end{aligned} \tag{3}$$

A node constraint may be always satisfied ($\top$), reference another node shape with name s_N that must be satisfied, require a specific node identity n in this place or require a node label l_N. It may also be the negation $\neg\phi_N$ or conjunction $\phi_N^1 \wedge \phi_N^2$ of other node constraints. Furthermore, the constraint $\geqslant_i p.\phi_N$ requires i nodes that can be reached via path p to conform to ϕ_N. $\odot(p_1, p_2)$ is an arbitrary comparison operation between sets of node identities that can be reached via the two path expressions p_1 and p_2.

Example 4. Consider the shape ${}_N\langle s_1, \geqslant_1 colleagueOf.Person, Employee\rangle$ targeting all nodes with the label *Employee*. s_1 requires at least one path *colleagueOf*, i.e., an outgoing edge that has the label *colleagueOf*, to a node which has the label *Person*. For the graph in Fig. 1, node 102 satisfies this constraint, because the only node reachable via path *colleagueOf* is node 100, and $Person \in \lambda(100)$. With analogous reasoning, the constraint does not hold for node 100, because $Person \notin \lambda(102)$

The aforementioned constraints were essentially transferred from core constraint components of the SHACL language. Novel kinds of constraints are printed in blue font. A qualified number restriction $\geqslant_i k.f$ restricts the number of values matching the predicate f for the property k. The qualified number constraints $\geqslant_i^{\leftarrow} \phi_E$ and $\geqslant_i^{\rightarrow} \phi_E$ require i incoming or outgoing edges that conform to the given edge constraint ϕ_E (defined below). $\odot(p_1, k_1, p_2, k_2)$ compares the annotated sets of values for properties k_1 and k_2, reached via paths p_1 and p_2 and $\odot(k_1, k_2)$ does the same for the current node.

Example 5. Consider the shape ${}_N\langle s_2, \geqslant_2 role.\texttt{string} \wedge s_1, name :$ "Gareth Keenan"$\rangle$, which targets all nodes n where "Gareth Keenan" $\in \sigma(n, name)$. For the graph G_{office} in Fig. 1, node 102 is the only target. The constraint $\geqslant_2 role.\texttt{string} \wedge s_1$ requires that this node conforms to shape s_1 from Example 4, as well as that the *role* property has at least two elements of type `string`. From Example 4 it follows that 102 conforms to s_1. The property $\sigma(102, role)$ has two values {"sales","team leader"}, both of which are strings. Therefore, node 102 conforms to s_2. Since node 102 is the only target of s_2, G_{office} conforms to s_2 as well.

Edge shapes apply to edges and, just as a node shape, require specific labels or properties for all targeted edges. Similarly to how node shapes constrain outgoing and incoming edges, edge shapes may constrain the source or destination node of an edge.

An edge shape is a tuple ${}_E\langle s_E, \phi_E, q_E\rangle$ consisting of shape name $s_E \in \text{Names}(S_E)$, an edge constraint ϕ_E and a target edge query q_E. Edge target queries are defined analogously to node target queries in Eq. (4) and Fig. 5.

$$q_E ::= \bot \mid e \mid l_E \mid k \mid k : v \tag{4}$$

Most constraint components of edge constraints ϕ_E are defined similarly to node constraints ϕ_N, albeit in terms of the respective edge identities e, edge labels l_E and edge shapes s_E. Unique to edge constraints are the constraints $\Leftarrow \phi_N$ and $\Rightarrow \phi_N$, which constrain source or destination nodes of an edge to conform to a node shape ϕ_N. The syntax of edge constraints ϕ_E is defined as in Eq. (5).

$$\phi_E ::= \top \mid s_E \mid e \mid l_E \mid \neg\phi_E \mid \phi_E^1 \wedge \phi_E^2 \mid \geqslant_i k.f \mid \Rightarrow \phi_N \mid \Leftarrow \phi_N \mid \odot(k_1, k_2) \tag{5}$$

$$\begin{aligned}
[\![\bot]\!]_G &= \emptyset \\
[\![e]\!]_G &= \{e\} \\
[\![l_E]\!]_G &= \{e \mid e \in E \wedge l_E \in \lambda(e)\} \\
[\![k]\!]_G &= \{e \mid e \in E \wedge \sigma(e,k) \neq \emptyset\} \\
[\![k : v]\!]_G &= \{e \mid e \in E \wedge v \in \sigma(e,k)\}
\end{aligned}$$

Fig. 5. Evaluation of target edge queries.

Example 6. Consider ${}_E\langle s_3, \Leftarrow \mathit{Person} \wedge \geqslant_1 \mathit{since}.(\geq 01/01/2020), \mathit{worksFor}\rangle$ which targets edges with the label *worksFor*. For the two matching edges of graph G_{office} in Fig. 1, 200 and 203, only 200 fulfils the constraint $\Leftarrow \mathit{Person}$, since $\mathit{Person} \in \lambda(100)$ and $\rho(200) = (100, 101)$. That is, the source node of edge 200 has the label *Person*. Only edge 203 fulfils the constraint $\geqslant_1 \mathit{since}. \geq 01/01/2020$, because at least one element of $\sigma(203, \mathit{since}) = \{02/08/2020\}$ fulfils the given value predicate $\geq 01/01/2020$, because $02/08/2020 \geq 01/01/2020$. Neither edge fulfils s_3.

Example 7. There is a difference between a node constraint $\geqslant_3 \mathit{colleagueOf}.\mathit{Person}$ and a node constraint $\geqslant_3^{\rightarrow} (\mathit{colleagueOf} \wedge \Rightarrow \mathit{Person})$. In the first case, we require 3 distinct nodes with the label *Person*, reachable via edges that match *colleagueOf*. In the second case, we require 3 outgoing edges labelled *colleagueOf* with destination nodes labelled *Person*. The nodes in the second case are not required to be distinct. Indeed, a graph with a single node having three self-loops could potentially fulfil the second, but never the first constraint.

In addition to these core constraints, we define useful syntactic sugar for both node constraints ϕ_N and edge constraint ϕ_E as shown in Fig. 6. For target queries, both conjunction and disjunction can also be defined as syntactic sugar (we use ϕ and q to mean either a node or edge constraint and query, respectively). Any shape with target $q_1 \wedge q_2$ and constraint ϕ is equivalent to a shape with target q_1 and the constraint $(\phi \wedge \phi_{q_2}) \vee \neg\phi_{q_2}$, where ϕ_{q_2} is the constraint equivalent to the target query (i.e., validating exactly the targets). Any shape s with target $q_1 \vee q_2$ and constraint ϕ can be expressed via two utility shapes with target q_1 and constraint s and target q_2 and constraint s, as well as the shape s with target $\bot$ and constraint ϕ.

3.3 Shape Semantics

Our definition of ProGS allows shape names to occur in constraints, meaning recursive cycles of references to other shapes can arise. Therefore, we follow an approach defined for recursive SHACL [6] and define evaluation of shapes on the basis of *partial assignments* for graph nodes and edges to sets of shapes. Our approach then relies on validating a given assignment in polynomial time (e.g., by guessing an assignment).

We formally define assignments on the basis of atoms, such that for each atom that pairs the name of a node shape with a node $s_N(n)$ or the name of an edge shape with an edge $s_E(e)$ a truth value from $\{0, 0.5, 1\}$ may be assigned.

$$\begin{array}{rlrl}
\bot & := \neg\top & \exists^{\leftarrow}\phi_E & := \geqslant_1^{\leftarrow} \phi_E \\
\leqslant_i^{\leftarrow} \phi_E & := \neg \geqslant_{i+1}^{\leftarrow} \phi_E & \exists p.\phi_N & := \geqslant_1 p.\phi_N \\
\leqslant_i p.\phi_N & := \neg \geqslant_{i+1} p.\phi_N & \exists k.f & := \geqslant_1 k.f \\
\leqslant_i k.f & := \neg \geqslant_{i+1} k.f & \forall^{\leftarrow}\phi_E & := \leqslant_0^{\leftarrow} \neg\phi_E \\
=_i^{\leftarrow} \phi_E & := \geqslant_i^{\leftarrow} \phi_E \wedge \leqslant_i^{\leftarrow} \phi_E & \forall p.\phi_N & := \leqslant_0 p.\neg\phi_N \\
=_i p.\phi_N & := \geqslant_i p.\phi_N \wedge \leqslant_i p.\phi_N & \forall k.f & := \leqslant_0 k.\neg f \\
=_i k.f & := \geqslant_i k.f \wedge \leqslant_i k.f & \phi_1 \vee \phi_2 & := \neg(\neg\phi_1 \wedge \neg\phi_2)
\end{array}$$

Fig. 6. Syntactic sugar for constraints, where ϕ is placeholder for either ϕ_N or ϕ_E. Definitions for syntactic sugar related to $\geqslant_i^{\rightarrow} \phi_E$ are omitted, since they are analogous to $\geqslant_i^{\leftarrow} \phi_E$.

Definition 2 (Atoms). *For a property graph $G = (N, E, \rho, \lambda, \sigma)$ and a set of shapes $S = S_N \cup S_E$, the set* $\text{atoms}(G, S) = \text{atoms}_N(G, S_N) \cup \text{atoms}_E(G, S_E)$ *where* $\text{atoms}_N(G, S_N) = \{s_N(n) \mid s_N \in S_N \wedge n \in N\}$ *and* $\text{atoms}_E(G, S_N) = \{s_E(e) \mid s_E \in S_E \wedge e \in E\}$ *is called the set of atoms of G and S.*

For the set of atoms of G and S, meaning essentially all tuples of shapes in S and nodes (or edges, respectively) in G, we define a partial assignment as a function Σ that maps for $x \in N \cup E$ all atoms $s(x) \in \text{atoms}(G, S)$ to 1, if the shape s is assigned to x, to 0 if $\neg s$ is assigned to x, and to 0.5 otherwise.

$$
\begin{aligned}
[\![\top]\!]^{\Sigma,n,G} &= 1\\
[\![s_N]\!]^{\Sigma,n,G} &= \Sigma(s_N(n))\\
[\![n']\!]^{\Sigma,n,G} &= [\,n' = n\,]\\
[\![l_N]\!]^{\Sigma,n,G} &= [\,l_N \in \lambda(n)\,]\\
[\![\neg\phi_N]\!]^{\Sigma,n,G} &= 1 - [\![\phi_N]\!]^{\Sigma,n,G}\\
[\![\phi_N^1 \wedge \phi_N^2]\!]^{\Sigma,n,G} &= \min\{[\![\phi_N^1]\!]^{\Sigma,n,G}, [\![\phi_N^2]\!]^{\Sigma,n,G}\}\\
[\![\geqslant_i p.\phi_N]\!]^{\Sigma,n,G} &= \begin{cases} 1 & |\{n_2 \mid n_2 \in [\![p]\!]^{\Sigma,n,G} \wedge [\![\phi_N]\!]^{\Sigma,n_2,G} = 1\}| \geq i\\ 0 & |[\![p]\!]^{\Sigma,n,G}| - \\ & |\{n_2 \mid n_2 \in [\![p]\!]^{\Sigma,n,G} \wedge [\![\phi_N]\!]^{\Sigma,n_2,G} = 0\}| < i\\ 0.5 & \text{otherwise}\end{cases}\\
[\![\odot\,(p_1, p_2)]\!]^{\Sigma,n,G} &= [\,[\![p_1]\!]^{\Sigma,n,G} \odot [\![p_2]\!]^{\Sigma,n,G}\,]\\
[\![\geqslant_i^{\leftarrow} \phi_E]\!]^{\Sigma,n,G} &= \begin{cases} 1 & |\{e \mid e \in E \wedge n_2 \in N \wedge \rho(e) = (n_2, n)\\ & \quad \wedge [\![\phi_E]\!]^{\Sigma,e,G} = 1\}| \geq i\\ 0 & |\{e \mid e \in E \wedge n_2 \in N \wedge \rho(e) = (n_2, n)\}| - \\ & |\{e \mid e \in E \wedge n_2 \in N \wedge \rho(e) = (n_2, n)\\ & \quad \wedge [\![\phi_E]\!]^{\Sigma,e,G} = 0\}| < i\\ 0.5 & \text{otherwise}\end{cases}\\
[\![\geqslant_i^{\rightarrow} \phi_E]\!]^{\Sigma,n,G} &= \begin{cases} 1 & |\{e \mid e \in E \wedge n_2 \in N \wedge \rho(e) = (n, n_2)\\ & \quad \wedge [\![\phi_E]\!]^{\Sigma,e,G} = 1\}| \geq i\\ 0 & |\{e \mid e \in E \wedge n_2 \in N \wedge \rho(e) = (n, n_2)\}| - \\ & |\{e \mid e \in E \wedge n_2 \in N \wedge \rho(e) = (n, n_2)\\ & \quad \wedge [\![\phi_E]\!]^{\Sigma,e,G} = 0\}| < i\\ 0.5 & \text{otherwise}\end{cases}\\
[\![\geqslant_i k.f]\!]^{\Sigma,n,G} &= [\,|\{v \mid v \in \sigma(n,k) \wedge f(v)\}| \geq i\,]\\
[\![\odot\,(p_1, k_1, p_2, k_2)]\!]^{\Sigma,n,G} &= \begin{array}{l}[\,\{v \mid n \in [\![p_1]\!]^{\Sigma,n,G}, v \in \sigma(n, k_1)\}\\ \quad \odot \{v \mid n \in [\![p_2]\!]^{\Sigma,n,G}, v \in \sigma(n, k_2)\}\,]\end{array}\\
[\![\odot\,(k_1, k_2)]\!]^{\Sigma,n,G} &= [\,\sigma(n, k_1) \odot \sigma(n, k_2)\,]
\end{aligned}
$$

Fig. 7. Evaluation rules for node constraints over graph G with assignment Σ.

Definition 3 (Partial Assignment). *Let G be a property graph and S a set of shapes. A partial assignment Σ is a total function $\Sigma : \text{atoms}(G, S) \rightarrow \{0, 0.5, 1\}$.*

Evaluating whether a node $n \in N$ of G satisfies a constraint ϕ_N, written $[\![\phi_N]\!]^{\Sigma,n,G}$ is defined in Fig. 7 and evaluating whether an edge $e \in E$ of G satisfies a constraint ϕ_E, written $[\![\phi_E]\!]^{\Sigma,e,G}$, is defined in Fig. 8. In the latter figure we omit cases that are trivially analogous to node shapes. In both figures, $[P]$ is similar to the Iverson bracket, such that $[P]$ evaluates to 1 (the constraint is satisfied) if P is true and 0 (the constraint is not satisfied) if P is false. Conditions for evaluation to 0.5 are given explicitly.

$$
\begin{aligned}
[\![s_E]\!]^{\Sigma,e,G} &= \Sigma(s_E(e)) \\
[\![e']\!]^{\Sigma,e,G} &= [\, e' = e \,] \\
[\![l_E]\!]^{\Sigma,e,G} &= [\, l_E \in \lambda(e) \,] \\
[\![\Rightarrow \phi_N]\!]^{\Sigma,e,G} &= [\![\phi_N]\!]^{\Sigma,n_2,G} \text{ where } (n_1, n_2) = \rho(e) \\
[\![\Leftarrow \phi_N]\!]^{\Sigma,e,G} &= [\![\phi_N]\!]^{\Sigma,n_1,G} \text{ where } (n_1, n_2) = \rho(e)
\end{aligned}
$$

Fig. 8. Evaluation rules for edge constraints over graph G with assignment Σ (omitting some cases that are analogous to cases in Fig. 7).

$$
\begin{aligned}
[\![l_E]\!]^{\Sigma,n,G} &= \{n_1 \mid e \in E \wedge (n, n_1) = \rho(e) \wedge l_E \in \lambda(e)\} \\
[\![p^-]\!]^{\Sigma,n,G} &= \{n_2 \mid n \in [\![p]\!]^{\Sigma,n_2,G}\} \\
[\![p_1/p_2]\!]^{\Sigma,n,G} &= \bigcup\{[\![p_2]\!]^{\Sigma,n_1,G} \mid n_1 \in [\![p_1]\!]^{\Sigma,n,G}\} \\
[\![p_1||p_2]\!]^{\Sigma,n,G} &= [\![p_1]\!]^{\Sigma,n,G} \cup [\![p_2]\!]^{\Sigma,n,G} \\
[\![p+]\!]^{\Sigma,n,G} &= \begin{cases} \emptyset, & \text{if } [\![p]\!]^{\Sigma,n,G} = \emptyset \\ [\![p]\!]^{\Sigma,n,G} \cup [\![p/p+]\!]^{\Sigma,n,G}, & \text{otherwise} \end{cases} \\
[\![p*]\!]^{\Sigma,n,G} &= \{n\} \cup [\![p+]\!]^{\Sigma,n,G} \\
[\![?p]\!]^{\Sigma,n,G} &= \{n\} \cup [\![p]\!]^{\Sigma,n,G}
\end{aligned}
$$

Fig. 9. Evaluation of path expressions.

The semantics of path expressions are defined in Fig. 9. We write $\{n_1, \ldots, n_i\} = [\![p]\!]^{\Sigma,n,G}$ for the evaluation of path p on graph G, such that nodes $n_1, \ldots, n_i$ can be reached via p from node n.

In order for a property graph G to be valid with respect to a set of shapes S, an assignment must exists which complies with all targets and constraints in S. Transferring terminology from [6] we call such an assignment *strictly faithful*.

Definition 4 (Strictly Faithful Assignment). *An assignment Σ for a property graph $G = (N, E, \rho, \lambda, \sigma)$ and a set of shapes S is strictly faithful, if and only if the following 4 properties hold (given shapes of the form ${}_N\langle s_N, \phi_N, q_N\rangle$ and ${}_E\langle s_E, \phi_E, q_E\rangle$):*

1. $\forall\ s_N(n) \in \mathit{atoms}(G, S) : \Sigma(s_N(n)) = [\![\phi_N]\!]^{\Sigma,n,G}$
2. $\forall\ s_E(e) \in \mathit{atoms}(G, S) : \Sigma(s_E(e)) = [\![\phi_E]\!]^{\Sigma,e,G}$
3. $\forall n \in [\![q_N]\!]_G : \Sigma(s_N(n)) = 1$
4. $\forall e \in [\![q_E]\!]_G : \Sigma(s_E(e)) = 1$

This means a strictly faithful assignment is an assignment, where all atoms are assigned exactly the result of constraint evaluation, all targets $n \in [\![q_N]\!]_G$ are assigned the respective shape s_N, and all targets $e \in [\![q_E]\!]_G$ are assigned the respective shape s_E. We define conformance of a graph with respect to a set of shapes on the basis of faithful assignments.

Definition 5 (Conformance). *A property graph $G = (N, E, \rho, \lambda, \sigma)$ conforms to a set of shapes S if and only if there exists at least one assignment Σ for G and S that is strictly faithful.*

3.4 Fulfilment of Requirements and Relationship to SHACL

As visualized by the colour coding of our definitions, the syntax of ProGS is an extension of the $\mathcal{L}$ language formalization of SHACL [6]. There are some exceptions arising from the existence of edges that have identities in property graphs. In fulfilment of requirements R3 and R4, ProGS allows qualifying the number of outgoing and incoming edges as well as reachable nodes, whereas SHACL only needs to be concerned with reachable nodes via some path.

Node shapes in SHACL may target all subjects or objects of an RDF property via `targetSubjectsOf` and `targetObjectsOf` expressions. In ProGS, these target queries are not required. Instead, fulfilling requirements R1 and R2, as well as R8, ProGS allows targeting of edges directly with specialized edge shapes. The respective source and destination nodes can then be constrained in these shapes via $\Leftarrow \phi_N$ and $\Rightarrow \phi_n$, respectively.

Finally, the handling of RDF literals in SHACL differs from constraints dealing with property annotations on nodes (R5 and R7) in ProGS. In addition, ProGS allows validating property annotations on edges (R6), which do not exist in RDF.

4 Complexity

We analyse the complexity of validating a property graph against a set of ProGS shapes. Before we define the validation problem VALID through the notion of faithfulness of assignments, we simplify the definition of faithful assignments with respect to target queries, by showing that it suffices to consider only cases where there is exactly one target node.

Proposition 1. *For a graph $G = (N, E, \rho, \lambda, \sigma)$ and a set of shapes $S = S_N \cup S_E$ with target nodes $n \in N$ for each $s_N \in S_N$ and target edges $e \in E$ for each $s_E \in S_E$, a graph G' and set of shapes S' can be constructed in linear time, such that G is valid against S if and only if G' is valid against S' and S' has a single target in G'.*

Proof (Sketch). Essentially, we construct edges from a new, single target node n_0 to previous target nodes and source nodes of target edges. Then we adapt constraints appropriately. Let $s_N^1, \ldots, s_N^n$ and $s_E^1, \ldots, s_E^n$ be shapes in S with targets $n_1^1, \ldots, n_1^m, \ldots, n_n^1, \ldots, n_n^m$ and targets $e_1^1, \ldots, e_1^m, \ldots, e_n^1, \ldots, e_n^m$. Extend G with a fresh node n_0 and fresh edges ne_i^j with $\rho(ne_i^j) = (n_0, n_i^j)$ for each target n_i^j as well as edges ee_i^j with $\rho(ee_i^j) = (n_0, n_1)$ where $(n_1, n_2) = \rho(e_i^j)$ for each target e_i^j. Then set all target queries for shapes in S to $\bot$ and introduce node shape s_{N_0} with target n_0 and constraint $\phi_{N_0} = \geqslant_1 ne_1^1.\phi_{s_N^1} \wedge \ldots \wedge \geqslant_1 ne_n^m.\phi_{s_N^n} \wedge \geqslant_1 ee_1^1. \geqslant_1 (e_1^1 \wedge \phi_{s_E^1}) \wedge \ldots \wedge \geqslant_1 ee_n^m. \geqslant_1 (e_n^m \wedge \phi_{s_E^n})$.

□

On the basis of this transformation, we can redefine strictly faithful assignments.

Definition 6 (Strictly Faithful Assignment for Graphs with a Single Target Node). *Let s_{N_0} be the shape and n_0 the node constructed by Proposition 1 as the single target node. An assignment Σ for a graph $G = (N, E, \rho, \lambda, \sigma)$ and a set of shapes S is strictly faithful, if and only if:*

1. $\forall\ s_N(n) \in atoms(G, S) : \Sigma(s_N(n)) = [\![\phi_N]\!]^{\Sigma,n,G}$
2. $\forall\ s_E(e) \in atoms(G, S) : \Sigma(s_E(e)) = [\![\phi_E]\!]^{\Sigma,e,G}$
3. $\Sigma(s_{N_0}(n_0)) = 1$

The validation problem VALID for validation of property graphs with respect to a set of ProGS shapes is defined as follows.

Definition 7 (Validation). *The problem of validating a property graph G with respect to a set of shapes S (such that in S there is exactly one shape s_{N_0} with a target query different from $\perp$ that targets node n_0, which can be constructed via Proposition 1 for any graph and set of shapes) is defined as* $\mathrm{VALID}(G, S, s_{N_0}(n_0))$.

We first show that VALID is in NP.

Theorem 1. *VALID is in NP.*

Proof (Sketch). In order to show that $\mathrm{VALID}(G, S, s_{N_0}(n_0))$ is in NP, we first construct, in polynomial time, an instance $\mathrm{VALID}(G', S', s_{N_0}(n_0))$ which is true if and only if $\mathrm{VALID}(G, S, s_{N_0}(n_0))$ is true, and S' does not contain any path expressions (except for l_E) and each constraint in S' has at most one operator. We assume an oracle for a strictly faithful assignment of such an instance $\mathrm{VALID}(G', S', s_{N_0}(n_0))$. Then we can, for each $s \in S'$, compute $[\![\phi_S]\!]^{\Sigma,n,G}$ for each $n \in N$ and $[\![\phi_s]\!]^{\Sigma,e,G}$ for each $e \in E$ in polynomial time in $|\Sigma| + |G'| + |S'|$. □

The complete proof can be found in an extended version of this work[1]. We next derive NP-hardness from the NP-hardness of $\mathcal{L}$.

Corollary 1. *RDF graph validation with $\mathcal{L}$, which is equivalent to SHACL, is clearly reducible to ProGS validation over property graphs, since RDF graphs can be trivially represented in property graphs and constraints in $\mathcal{L}$ are a subset of ProGS constraints. According to [6], $\mathcal{L}$ is NP-hard. Therefore, ProGS is also NP-hard.*

Then we can also conclude that VALID for ProGS is NP-complete.

Corollary 2. *VALID is NP-complete, since it is both NP-hard (shown in Corollary 1) and in NP (shown in Theorem 1).*

We only consider the combined complexity here, even though graphs are typically significantly larger than sets of shapes. However, from this we infer that validation for a fixed set of shapes (data complexity) or a fixed graph (constraint complexity) are also NP-complete, since they are already NP-complete for $\mathcal{L}$ as shown in [6], and combined complexity of validation for ProGS is in NP.

[1] https://arxiv.org/abs/2107.05566.

5 Implementation

Drawing inspiration from an experimental feature of the SHaclEX [24] implementation of ShEx [23] and SHACL [22], we implement a prototypical validator for ProGS by encoding the validation problem as an answer set program. Answer set programming (ASP) allows for declarative implementations of NP-hard search problems, such as ProGS validation with faithful assignments. In particular, we rely on ASP for efficiently finding candidate assignments (in the worst-case considering all possible assignments), while deciding whether an assignment is faithful is a straightforward mapping of our validation semantics into another ASP model.

The implementation consists of three components: An encoding of property graphs and ProGS shapes, both of which are straight-forward and can be generated from non-ASP representations. A set of rules directly representing the validation semantics of ProGS (Sect. 3.3). And finally the search problem of finding faithful assignments. With these components, an ASP solver (our implementation relies on Clingo[2]) produces one (or more) faithful assignments for the graph and set of shapes (if any exist).

In addition to the ASP encoding, we also provide a surrounding set of tools, including a concrete syntax for ProGS shapes and a corresponding parser, as well as a tool for extracting and encoding Neo4j[3] instances. The graph encoding is based on the Neo4j JSON export format and therefore straight-forward to replicate for other property-graph stores. The tool suite is available on GitHub[4], including further documentation and examples. More details about the ASP encoding and a short demonstration can be found in the extended version of this work.

5.1 Towards Practical Implementations of ProGS

Our implementation is well-suited as a reference implementation, for experimenting with ProGS examples, and for validating smaller-sized graphs. For large-scale graphs, the explicit ASP encoding of the data graph may be too inefficient, both in terms of runtime and memory requirements. Instead, efficient validation demands an implementation operating directly on a specific property-graph store. Such an implementation could, for example, aim to replicate the resolution approach of an ASP solver for finding candidate assignments and evaluate the validation procedure directly on the graph. For simplified SHACL shapes that do not include recursive shape references, efficient validation approaches are well-known and widely used in real-world SHACL implementations. These approaches, operating on graph stores directly, could be applied for ProGS as well. Another alternative would be to adapt validation over SPARQL endpoints [5] for Cypher and ProGS instead. Indeed, neosemantics [15] relies on

[2] https://potassco.org/clingo/.
[3] https://neo4j.com/.
[4] https://github.com/softlang/progs.

Cypher for the validation of SHACL over RDF graphs encoded as property graphs. Such an approach, as is also shown by [5], can be extended to validate recursive shapes by inclusion of a SAT solver.

6 Related Work

There are a number of schema languages for property graphs in proprietary implementations of graph databases. For instance, the data definition language for Cypher [9] described in the Neo4j manual [14] allows for simple constraints regarding the existence or uniqueness of properties. For TigerGraph [7], a similar implementation exists. However, these systems lack a formal description, making their expressiveness, features and complexity hard to assess.

Only a small number of property-graph schema languages have been formally defined. In [11], the GraphQL [8] schema language is used to define restrictive property-graph schemas, where for each node label a GraphQL object type can be defined. This allows for constraining the existence of certain properties, edges, and properties on these edges via field definitions of the object types. The schemas are closely tied to node labels, meaning the approach does not allow for the validation of edges as individual entities, which is crucial for validating metadata annotations across an entire graph. The approach also omits other elements supported by ProGS, such as negation, qualified number restrictions and path expressions in number restrictions or equality constraints. Validation with constraints that are associated to labels can be emulated with ProGS target queries. Graph validation based on GraphQL was shown to be in AC_0.

[1] defines property graph schemas, also focusing on node and edge types on the basis of labels. In particular, schemas allow for restricting the data types of specific properties on nodes and edges, as well as the edges allowed between node types. More advanced constraints are mentioned, but not formally defined. In general, this approach only provides a small subset of the features of ProGS.

While shape-based validation approaches such as SHACL [22] and ShEx [23] exist for validating RDF graphs, to the best of our knowledge no shape-based validation language for property graphs has been formally defined until now. A syntactic construct for SHACL validation of RDF* (and other reification-based RDF extensions) has been proposed in an unofficial draft proposal [13], though no semantics has been specified. The `reifiableBy` construct allows constraining an edge via a node shape for provenance annotations. The approach is similar to our notion of edge shapes and our semantics can be applied, as long as graphs are restricted to property graphs (i.e., edge properties are restricted to a given set of value domains). Finally, there exists an extension for Neo4j which implements SHACL validation for RDF subsets of property graphs [15].

7 Concluding Remarks

We present ProGS, a shape language extending SHACL for validating property graphs. We define the semantics of this language based on the notion of faithfulness of partial assignments and are therefore able to support shape references and negation. Despite the addition of property-graph specific constructs, such as edge shapes that target edges with identities, the complexity of validating graphs against sets of ProGS shapes does not increase when compared to SHACL. The validation problem remains NP-complete.

As future work, we plan to investigate the satisfiability problem of ProGS shapes and then further utilize these results to define a validation approach for property-graph queries. We are also interested in extending ProGS with the unique features introduced by G-CORE, in particular first-class paths, and RDF*, in particular triples in object position of other triples.

References

1. Angles, R.: The property graph database model. In: Proceedings of International Workshop on Foundations of Data Management. CEUR, vol. 2100 (2018). http://ceur-ws.org/Vol-2100/paper26.pdf
2. Angles, R., et al.: G-CORE: a core for future graph query languages. In: Proceedings of SIGMOD, pp. 1421–1432. ACM (2018). https://doi.org/10.1145/3183713.3190654
3. Angles, R., Thakkar, H., Tomaszuk, D.: Mapping RDF databases to property graph databases. IEEE Access **8**, 86091–86110 (2020). https://doi.org/10.1109/ACCESS.2020.2993117
4. Apache: Gremlin Property Graph Model (2016). https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model
5. Corman, J., Florenzano, F., Reutter, J.L., Savković, O.: Validating SHACL constraints over a SPARQL endpoint. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11778, pp. 145–163. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30793-6_9
6. Corman, J., Reutter, J.L., Savković, O.: Semantics and validation of recursive SHACL. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11136, pp. 318–336. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00671-6_19
7. Deutsch, A., Xu, Y., Wu, M., Lee, V.E.: TigerGraph: a native MPP graph database. CoRR abs/1901.08248 (2019)
8. Facebook: GraphQL Spec. (2018). https://graphql.github.io/graphql-spec/
9. Francis, N., et al.: Cypher: an evolving query language for property graphs. In: Proceedings of SIGMOD, pp. 1433–1445. ACM (2018). https://doi.org/10.1145/3183713.3190657
10. Hartig, O.: Rdf* and sparql*: an alternative approach to annotate statements in RDF. In: Proceedings of ISWC, Posters & Demonstrations and Industry Tracks. CEUR Workshop Proceedings, vol. 1963. CEUR-WS.org (2017). http://ceur-ws.org/Vol-1963/paper593.pdf
11. Hartig, O., Hidders, J.: Defining schemas for property graphs by using the graphql schema definition language. In: GRADES/NDA@SIGMOD/PODS, pp. 6:1–6:11. ACM (2019). https://doi.org/10.1145/3327964.3328495
12. ISO/IEC JTC1 SC32 WG3: GQL Standardization Project (2020). https://www.gqlstandards.org/
13. Knublauch, H.: DASH Reification Support for SHACL (2021). http://datashapes.org/reification.html
14. Neo4j: Neo4j Constraints (2020). https://neo4j.com/docs/cypher-manual/4.2/administration/constraints/

15. Neosemantics: Neo4j Neosemantics Validation (2020). https://neo4j.com/labs/neosemantics/4.0/validation/
16. openCypher: openCypher Project (2020). http://www.opencypher.org/
17. Oracle: PGQL 1.3 Specification (2020). https://pgql-lang.org/spec/1.3/
18. Seifer, P., Härtel, J., Leinberger, M., Lämmel, R., Staab, S.: Empirical study on the usage of graph query languages in open source Java projects. In: Proceedings of Software Language Engineering, pp. 152–166. ACM (2019). https://doi.org/10.1145/3357766.3359541
19. Udrea, O., Recupero, D.R., Subrahmanian, V.S.: Annotated RDF. ACM Trans. Comput. Log. **11**(2), 10:1–10:41 (2010). https://doi.org/10.1145/1656242.1656245
20. Vrandecic, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014). https://doi.org/10.1145/2629489
21. W3C: RDF Concepts and Abstract Syntax (2014). https://www.w3.org/TR/rdf11-concepts/
22. W3C: Shapes constraint language (SHACL) (2017). https://www.w3.org/TR/shacl/
23. W3C: Shapes expressions language (ShEx) (2019). http://shex.io/shex-semantics/
24. WESO: Shaclex (2021). https://github.com/weso/shaclex