SHACL et ShEx

January 31, 2025

Positive Rules in Knowledge Graphs

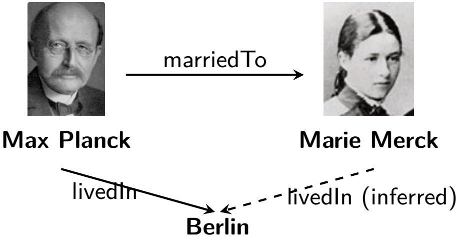
Goal of Rule Mining: Discover logical patterns in knowledge graphs.

Positive rules capture frequent patterns (valid relations).

Example Rule R**:**

$$\texttt{marriedTo(x,y), livedIn(x,z)} \Rightarrow \texttt{livedIn(y,z)}$$

Goal: Predict new facts based on knowledge grounded in $\mathcal{K}$.



Last week: discovery of the positive Horn Rules in Knowledge Graphs.

Evaluation Metrics

Rule Support

Definition: The support of a rule R is the number of true facts it generates:

$$support(R) = |\{p : (\mathcal{K} \wedge R \models p) \wedge p \in \mathcal{K}\}|$$

Property: The support decreases as a rule becomes more specialized.

Utility: A rule is not refined if its support falls below a threshold.

Confidence Measure of a Rule

Definition: The confidence of a rule R is the proportion of true predictions among all predictions:

$$confidence(R) = \frac{support(R)}{support(R) + |cex(R)|}$$

where $cex(R)$ represents the counterexamples of R.

- **Support:** Relevance of a rule.
- **Confidence:** Accuracy of a rule.

Counter Examples:

- **Closed World Assumption (CWA):** What is not known is false.
- **Open World Assumption (OWA):** What is not known is unknown (and could therefore be true).
- **Partial Completeness Assumption (PCA):** All or none of the relationships r are known for a subject s.

Analysis of the pca-conf metric (1)

Does the PCA metric effectively detect sub-properties?

- Example: **authorOf** $\sqsubseteq$ **createdBy**

$$\mathbf{authorOf}(x, y) \rightarrow \mathbf{createdBy}(x, y)$$

- Detecting sub-properties is a fundamental concept in Description Logic.
- It is also part of OWL recommendations.

Analysis of the pca-conf metric (2)

It is sufficient to annalyse the following formula:

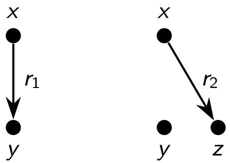
$$\mathit{pca\text{-}conf}(R) = \frac{\mathit{support}(R)}{\mathit{support}(R) + |\mathit{cex}(R)|} \quad \text{where } R = r_1 \sqsubseteq r_2$$

with $\mathit{support}(R) = \{(x, y) : r_1(x, y) \in \mathcal{K} \wedge r_2(x, y) \in \mathcal{K}\}$

and $\mathit{cex}(R) = \{(x, y) : r_1(x, y) \in \mathcal{K} \wedge r_2(x, y) \notin \mathcal{K} \wedge \exists z : r_2(x, z) \in \mathcal{K}\}$

Elements of $\boldsymbol{cex}(R)$: an illustration

$(x, y) \in \mathit{extra}(R)$



Surprising consequence: Both support and extra are defined only over the common domain of the two relations r_1 and r_2.

Implications for sub-property detection

- In addition to detecting that **authorOf** $\sqsubseteq$ **createdBy**, the *pca-conf* metric also assigns a maximum score to the inverse relation **createdBy** $\sqsubseteq$ **authorOf**.
- This is explained by the fact that both relations are equivalent over the common domain.
- This example satisfies the condition of Property 2, presented in the report, allowing this implication to be generalized.

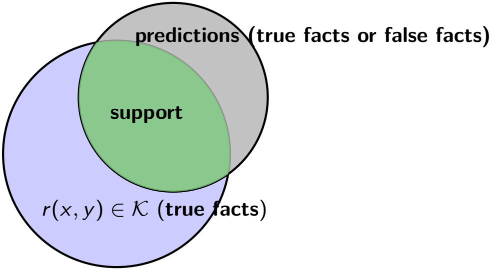
Head Coverage

Definition:

$$hc(\boldsymbol{B} \Rightarrow r(x,y)) = \frac{support(\boldsymbol{B} \Rightarrow r(x,y))}{|\{(x,y) \mid r(x,y) \in \mathcal{K}\}|}$$

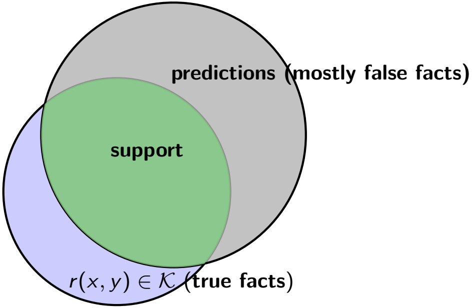
- The fraction of true occurrences of the head predicate captured by the rule.
- Normalized by the total occurrences of the head predicate r in the knowledge graph $\mathcal{K}$.

Intuition: Head Coverage (hc) captures how much of a given predicate's facts are **explained** by a rule.



Overfitting or the Lack of Generality

- Optimizing **only** for high **head coverage** (hc) may lead to rules that generate **many false predictions**.
- In other words, such rules **lack generality**.
  They fail to distinguish between correct and incorrect predictions.



Towards weighting in the coverage of the negative facts

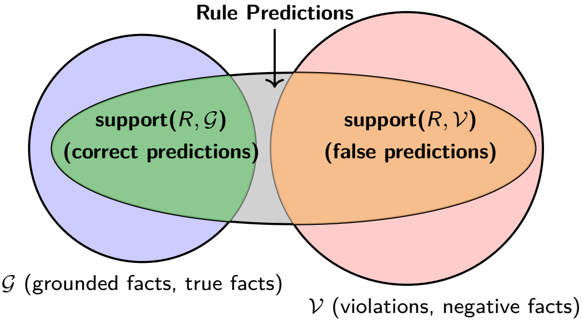
Intuition: A rule should be penalized when it supports negative facts.

Towards weighting in the coverage of the negative facts

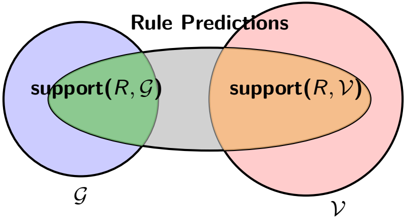
Intuition: A rule should be penalized when it supports negative facts.

Coverage of a given set $\mathcal{S}$ (Support of the rule in $\mathcal{S}$)

$$support(R, \mathcal{S}) = |\{p : (\mathcal{K} \wedge R \models p) \wedge p \in \mathcal{S}\}|$$

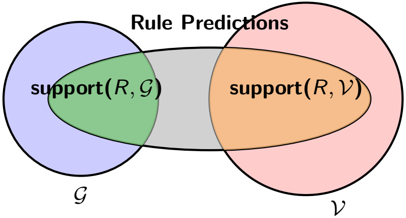


Metric: The Weight of a Rule R



Motivation: The two support measures must be normalized for fair comparison.

Metric: The Weight of a Rule R

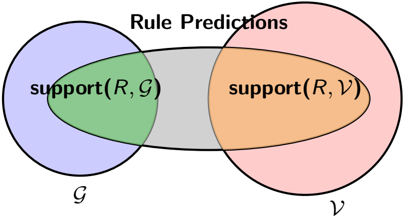


Motivation: The two support measures must be normalized for fair comparison.

A first attempt: To balance precision and recall, we propose:

$$w(R) = \alpha(1 - \frac{support(R, \mathcal{G})}{|\mathcal{G}|}) + \beta(\frac{support(R, \mathcal{V})}{|\mathcal{V}|})$$

Metric: The Weight of a Rule R



Motivation: The two support measures must be normalized for fair comparison.

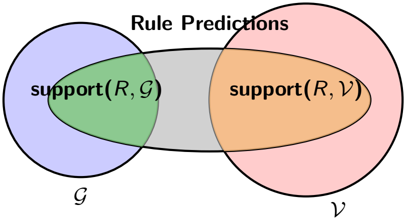
A first attempt: To balance precision and recall, we propose:

$$w(R) = \alpha\left(1 - \frac{support(R, \mathcal{G})}{|\mathcal{G}|}\right) + \beta\left(\frac{support(R, \mathcal{V})}{|\mathcal{V}|}\right)$$

Interpretation: The parameters α and β allow adjusting the trade-off.

- A higher β favors precision (penalizing rules covering negative facts).
- A higher α favors recall (rewarding rules that cover more true facts).

Metric: The Weight of a Rule R



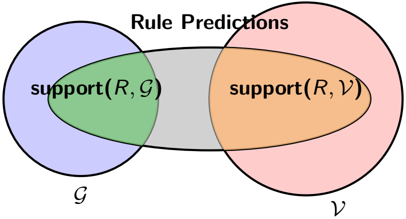
Motivation: The two support measures must be normalized for fair comparison.

A first attempt: To balance precision and recall, we propose:

$$w(R) = \alpha(1 - \frac{support(R, \mathcal{G})}{|\mathcal{G}|}) + \beta(\frac{support(R, \mathcal{V})}{|\mathcal{V}|})$$

Discussion: What might be problematic with this approach?

Metric: The Weight of a Rule R



Motivation: The two support measures must be normalized for fair comparison.

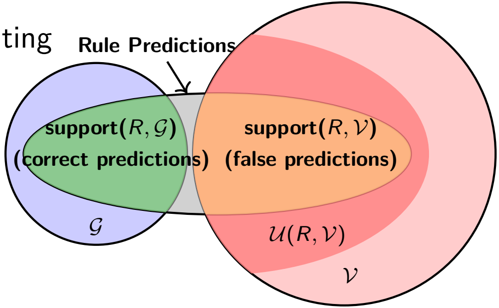
A first attempt: To balance precision and recall, we propose:

$$w(R) = \alpha\left(1 - \frac{support(R, \mathcal{G})}{|\mathcal{G}|}\right) + \beta\left(\frac{support(R, \mathcal{V})}{|\mathcal{V}|}\right)$$

Discussion: What might be problematic with this approach?

- The set $\mathcal{G}$ of grounded facts in the knowledge graph is **finite**.
- The set of negative facts is **infinite**—$\mathcal{V}$ can be arbitrarily large.

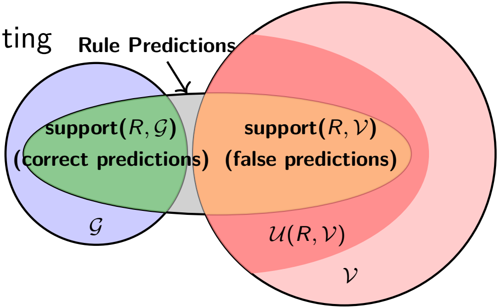
RuDiK: Metric for Rule Weighting



$$w(R) = \alpha \left(1 - \frac{support(R, \mathcal{G})}{|\mathcal{G}|}\right) + \beta \left(\frac{support(R, \mathcal{V})}{|\mathcal{U}(R, \mathcal{V})|}\right)$$

- The set $\mathcal{U}(R, \mathcal{V})$ is constructed such that $support(R, \mathcal{V}) \subseteq \mathcal{U}(R, \mathcal{V})$, ensuring a **normalized** value for the second fraction.

RuDiK: Metric for Rule Weighting

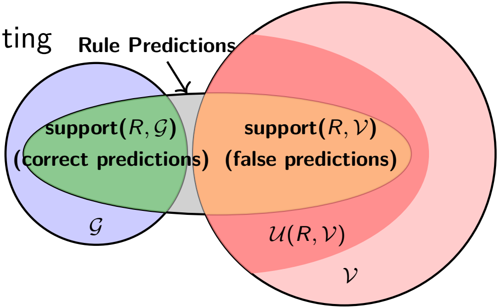


$$w(R) = \alpha \left(1 - \frac{support(R, \mathcal{G})}{|\mathcal{G}|}\right) + \beta \left(\frac{support(R, \mathcal{V})}{|\mathcal{U}(R, \mathcal{V})|}\right)$$

- The set $\mathcal{U}(R, \mathcal{V})$ is constructed such that $support(R, \mathcal{V}) \subseteq \mathcal{U}(R, \mathcal{V})$, ensuring a **normalized** value for the second fraction.
- Given the rule $R : \boldsymbol{B} \Rightarrow r(x, y)$, RuDiK lets

$$\mathcal{G} = |\{(x, y) \mid r(x, y) \in \mathcal{K}\}|$$

RuDiK: Metric for Rule Weighting



$$w(R) = \alpha \left(1 - \frac{support(R, \mathcal{G})}{|\mathcal{G}|}\right) + \beta \left(\frac{support(R, \mathcal{V})}{|\mathcal{U}(R, \mathcal{V})|}\right)$$

- The set $\mathcal{U}(R, \mathcal{V})$ is constructed such that $support(R, \mathcal{V}) \subseteq \mathcal{U}(R, \mathcal{V})$, ensuring a **normalized** value for the second fraction.
- Given the rule $R : \boldsymbol{B} \Rightarrow r(x, y)$, RuDiK lets

$$\mathcal{G} = |\{(x, y) \mid r(x, y) \in \mathcal{K}\}|$$

- The first fraction corresponds to the **head coverage** $hc(R)$.

RuDiK: General Optimization Problem

Input:

- r — the target predicate (e.g., spouse)
- **Generation set** $\mathcal{G}$ — positive examples
- **Validation set** $\mathcal{V}$ — negative examples

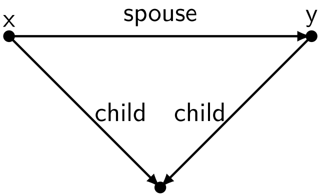
Output: A set of positive rules R that minimize $w(R)$.

$$w(R) = \alpha \left(1 - \frac{support(R, \mathcal{G})}{|\mathcal{G}|}\right) + \beta \left(\frac{support(R, \mathcal{V})}{|\mathcal{U}(R, \mathcal{V})|}\right)$$

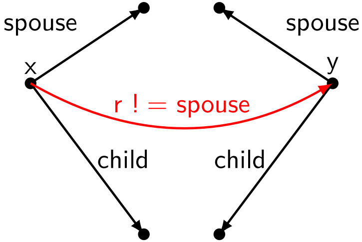
Optimization Goals:

- Maximizes the coverage of positive facts (i.e., increases recall $\uparrow$).
- Minimizes the coverage of negative facts (i.e., reduces false positives $\downarrow$).

Generating Negative Facts



Rule R

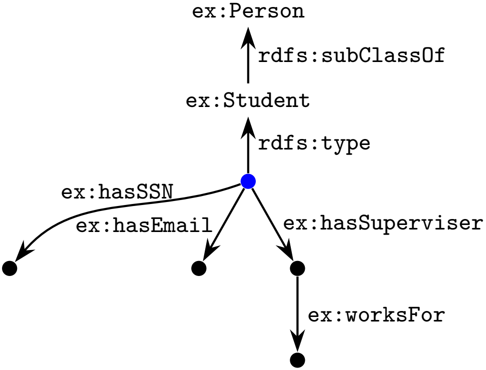


Rule R^- **for generating negative examples** (x, y)

To make the size of $\mathcal{U}(\mathcal{R}, \mathcal{V})$ comparable to $\mathcal{K}$, it is required that (x, y) are in a relation different from the target relation (spouse).

Validation Constraints in Practice

Validation Constraints



Example Constraints:

1. Each student must have exactly one Social Security Number (SSN).
2. The Social Security Number (SSN) must be unique.
3. The email of each student must be well-formed.
4. All supervisors must work for the same organization.

SHACL: Shape-Based Constraint Language

- SHACL is a standard language used to validate RDF graphs.
  It has been a W3C Recommendation since 2017.
- It defines **validation constraints** on classes, nodes, properties, and literals.
- Designed as a **schema language** for RDF graphs.
- SHACL constraints are expressed in the RDF language.

Shape Graphs and Shapes

SHACL Validation Process

Inputs:

- a **data graph**,
- a **shape graph**.

Output: a validation report.

Shape Graph

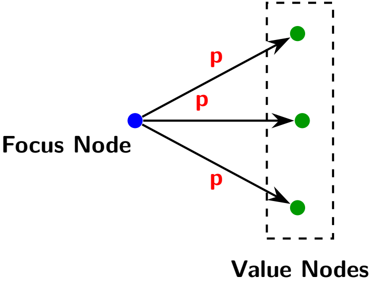
- A shape graph contains zero or more **shapes**.
- The shape graph is represented as an RDF entity.
- Each shape is also an RDF entity.
- All statements about these shapes are expressed as RDF triples.
- Predicates are defined by the vocabulary `http://www.w3.org/ns/shacl#`, typically aliased as `sh`.

Shapes

A shape specifies how to validate a **node** based on the values of its properties and other characteristics.

The core SHACL language defines two types of shapes:

- **Node Shape**: Defines constraints applied directly to a **focus node**.
- **Property Shape**: Defines constraints on the **value nodes**, i.e., the nodes reachable from the **focus node** of a Node Shape via a path **p**.



Node Shape

- A Node Shape is an instance of the class `sh:NodeShape`.
- It must not include any statement with `sh:path` as a predicate.
- It specifies the target nodes to validate using dedicated properties such as `sh:targetClass`, `sh:targetNode`, etc.

Targeting a Class:

```
ex:StudentShape
    a sh:NodeShape;
    sh:targetClass ex:Student.
```

Targeting Specific Nodes:

```
ex:StudentShape
    a sh:NodeShape;
    sh:targetNode ex:Alice, ex:Bob.
```

Node Shape: How to Specify Target Nodes?

Target Type	Description
`sh:targetClass` `c`	Targets all* instances of the class `c`.
`sh:targetNode` `list-of-nodes`	Targets the nodes specified in the list.
`sh:targetSubjectOf` `p`	Targets the subjects associated with the predicate `p`.
`sh:targetObjectOf` `p`	Targets the objects associated with the predicate `p`.

Note*: In SHACL, **x** is an instance of `c` if:

x rdf:type/rdfs:subClassOf* c

This holds only if the SHACL processor supports an RDFS inference regime or another regime specified via `sh:entailment`.

Remark: The capabilities of the SHACL processor may vary depending on its support for inference.

Implicit Targets for Classes

SHACL: If an entity is simultaneously defined as both a **class and a shape**[1], then all its instances in the data graph become target nodes for that shape.

Example Shape:

```
ex:Student a rdfs:Class;
           a sh:NodeShape.
```

Data Graph:

```
ex:Alice a ex:Student ;
         ex:hasName "Alice".

ex:NewYork a ex:Place.
```

✓ ✗

Target Node: `ex:Alice` is a target node for the shape `ex:Student` because it is an instance of the class `ex:Student`.

[1] This applies to both Node Shapes and Property Shapes.

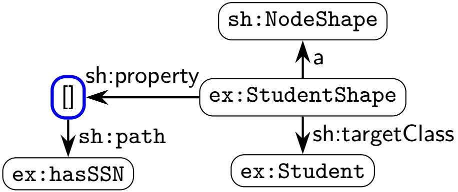
Introduction to Property Shapes

A Property Shape:

- Is associated with a Node Shape.
- Defines constraints on the **value nodes**, which are reachable from the **focus node** via a path `p`.
- Is typically represented by a blank RDF node.
- It is recommended to declare it as an instance of `sh:PropertyShape`.
- Specifies the path `p` via the `sh:path` property (unique for each shape).

Example:

```
ex:StudentShape
    a sh:NodeShape ;
    sh:targetClass ex:Student ;
    sh:property [
        a sh:PropertyShape
        sh:path ex:hasSSN ;
        sh:maxCount 1;
    ] .
```



- The Property Shape is [], a blank node according to RDF.

Property Shapes: Specifying the Path

```
property_shape sh:path path .
```

where `path` is a SHACL path belonging to one of the following categories:

Category	Example	Language
Direct Predicate	Unique IRI, similar to a SPARQL property.	
Inverse Path	`[sh:inversePath yago:parentOf]`	SHACL
	`^yago:parentOf`	SPARQL
Sequence Path	`(yago:parentOf rdfs:label)`	SHACL
	`yago:parentOf/rdfs:label`	SPARQL
Alternative Path	`[sh:alternativePath (f:friend f:knows)]`	SHACL
	`f:friend|f:knows`	SPARQL
Zero or One Path	`[sh:zeroOrOnePath yago:locatedIn]`	SHACL
	`yago:locatedIn?`	SPARQL
One or More Paths	`[sh:oneOrMorePath yago:locatedIn]`	SHACL
	`yago:locatedIn+`	SPARQL
Zero or More Paths	`([sh:zeroOrMorePath yago:locatedIn] rdfs:label)`	SHACL
	`yago:locatedIn*/rdfs:label`	SPARQL

Basic Constraints

Constraints applicable to both types of shapes:

- Nature of the target nodes.
- Numerical comparisons of node values.
- Length of the node representation.
- Conformity of nodes to a regular expression.
- Presence and handling of language tags.

Constraints specific to property shapes (value nodes):

- Cardinality of the set of value nodes.
- Membership of nodes in a set of values.
- Constraints on pairs of properties.

Constraint: Nature of Target Nodes (1)

```
propertyShape or nodeShape sh:class c .
```

- The nodes targeted by the shape must be instances of the class c.
- Multiple values for `sh:class` are interpreted as a conjunction.

Constraint: Nature of Target Nodes (2)

```
propertyShape or nodeShape sh:datatype t .
```

- The nodes targeted by the shape must be literals of type t.
- A shape can have at most one value for `sh:datatype`.
- To test if value nodes have a language tag, use `rdf:langString` as the datatype.

Constraint: Nature of Target Nodes (3)

```
propertyShape or nodeShape sh:nodeKind k .
```

- There can be at most one `sh:nodeKind` declaration.

k
sh:BlankNode
sh:IRI
sh:Literal
sh:BlankNodeOrIRI
sh:BlankNodeOrLiteral
sh:IRIOrLiteral

Numerical Constraint on Target Node Values

```
propertyShape or nodeShape comparison n .
```

comparison
sh:minInclusive
sh:minExclusive
sh:maxInclusive
sh:maxExclusive

- The constraint is that all targeted nodes must satisfy the comparison.
- Typically, the same nodes are also subject to a `sh:datatype` constraint with a numeric type.

Example Shape:

```
c:GradesForFrenchStudents
    a sh:NodeShape;
    sh:targetClass
        ex:FrenchStudent;
    sh:property [
        sh:path ex:grade;
        sh:minInclusive 0;
        sh:maxInclusive 20;
    ] .
```

Data:

```
ex:Paul a ex:FrenchStudent;
       ex:grade 15.

ex:Anne a ex:FrenchStudent;
       ex:grade 25.
```

✓

✗

Constraints: Length of Target Node Representations

```
propertyShape or nodeShape sh:minLength n ;
                           sh:maxLength  m .
```

- The target nodes must be either literals or IRIs.
- Each of the two specifications must be unique.

Constraint: Each Target Node Matches a Regular Expression

```
propertyShape or nodeShape sh:pattern regex ;
                           sh:flags  flag .
```

- regex is a REGEX expression.
- flag specifies whether the constraint is case-sensitive, e.g., "i" (W3C link).

Example Shape:

```
c:CSCourses a sh:NodeShape ;
   sh:targetClass p:Course ;
   sh:property [
      sh:path p:id;
      sh:pattern "^CS[0-9]{3}$";
   ] .
```

Data:

```
p:C1    a p:Course;
        p:id "CS101".

p:C2 a p:Course;
        p:id "CS50".

p:Course5 a p:Course ;
        p:id "MATH123".
```

✓

✗

✗

Constraint: Language Tags of Target Nodes

```
propertyShape or nodeShape sh:languageIn lang_list .
property_shape sh:uniqueLang true.
```

- The `sh:uniqueLang` constraint applies only to value nodes.
- `lang_list` specifies the allowed language tags for each target node.
- `sh:uniqueLang true` ensures there is at most one value per language tag.

Shape:

```
c:CourseShape a sh:NodeShape ;
   sh:targetClass p:Course ;
   sh:property [
    sh:path p:title;
    sh:languageIn ("fr", "en");
    sh:uniqueLang true;
   ] .
```

Data:

```
p:C1 a p:Course;
   p:title "Algorithmique"@fr;
   p:title "Algorithms"@en.

p:C3 a p:Course ;
   p:title "Probabilités"@fr;
   p:title "Probabilities"@fr.
```

✓

✗

Constraint: Cardinality of the Set of Value Nodes

```
propertyShape  sh:path p  ;
               sh:maxCount n  ;
               sh:minCount m  .
```

- n is the upper bound on the number of value nodes selected by p.
- m is the lower bound on the number of value nodes selected by p.
- The values n and m must be literals of type xsd:integer.
- Only one value is allowed for each of the properties.
- This constraint applies only to Property Shapes.

Constraint: Value Nodes Take a Given Value

```
propertyShape sh:path p;
              sh:hasValue v.
```

- All values selected by p must be equal to v.

Constraint: Belonging to a Set of Values

```
propertyShape sh:path p;
              sh:in (v1 v2 ... vn).
```

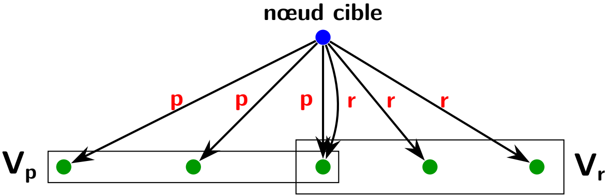
- Each value selected by `p` must be one of the values `v1, v2, ... vn`.

Constraint on Property Pairs

```
propertyShape sh:path p ;
              constraint r.
```

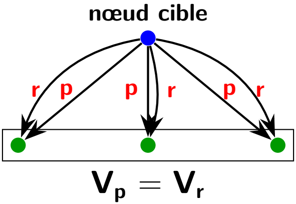
constraint
sh:equals
sh:disjoint
sh:lessThan
sh:lessThanOrEquals

- **p** is a path, and **r** is a property (relation).
- Let $\mathbf{V_p}$ be the set of value nodes selected by the path **p**.
- Let $\mathbf{V_r}$ be the set of value nodes selected by the predicate **r**.
- The constraints compare the sets $\mathbf{V_p}$ and $\mathbf{V_r}$.



`sh:equals`: Path and Predicate Select the Same Nodes

```
propertyShape sh:path p ;
               sh:equals r.
```



- $\mathbf{V_r} = \mathbf{V_p}$: The path `p` and the predicate `r` point to the same nodes.

Constraint: Predicate and Path Select the Same Nodes

Example:

```
c:PhDStudentShape a sh:NodeShape ;
    sh:targetClass p:PhDStudent ;
    sh:property [
        sh:path (p:doctoralAdviser p:affiliatedWith) ;
        sh:equals p:enrolledAt ;
    ] .
```

Data:

```
p:Pierre a p:Professor ;
        p:affiliatedWith p:IPP .
p:Anne a p:PhDStudent ;
        p:doctoralAdviser p:Pierre ;
        p:enrolledAt p:IPP .
p:Paul a p:PhDStudent ;
        p:doctoralAdviser p:Pierre ;
        p:enrolledAt p:UPS .
```

✓

✓

✗

Constraint: Predicate and Path Select the Same Nodes

Example: For a PhD student, the organizations their doctoral adviser is affiliated with must match the organizations they are enrolled in.

```
c:PhDStudentShape a sh:NodeShape ;
    sh:targetClass p:PhDStudent ;
    sh:property [
        sh:path (p:doctoralAdviser p:affiliatedWith) ;
        sh:equals p:enrolledAt ;
    ] .
```

Data:

```
p:Pierre a p:Professor ;
         p:affiliatedWith p:IPP .
p:Anne a p:PhDStudent ;
         p:doctoralAdviser p:Pierre ;
         p:enrolledAt p:IPP .
p:Paul a p:PhDStudent ;
         p:doctoralAdviser p:Pierre ;
         p:enrolledAt p:UPS .
```

✓

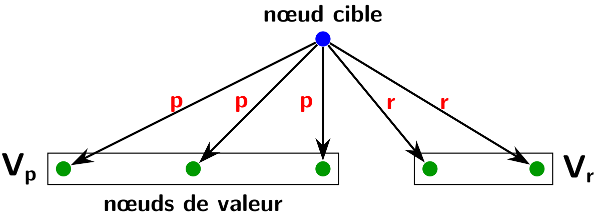
✓

✗

sh:lessThan

```
propertyShape sh:path p ;
              sh:lessThan r.
```

- The sh:lessThan constraint ensures that the values of $\mathbf{V_p}$ are strictly less than the values of $\mathbf{V_r}$.
- Formally: $\mathbf{max}(\mathbf{V_p}) < \mathbf{min}(\mathbf{V_r})$.



Logical Operators: sh:not

```
shape sh:not [ shapeToNegate ].
```

- The operator **sh:not** is used to negate the constraints defined by the shape specified inside it.
- Applicable to both types of shapes: nodeShape and propertyShape.

Logical Operators: `sh:and`, `sh:or`, `sh:xone`

```
shape operator (SHACL_list_of_shapes).
```

operator
`sh:and`
`sh:or`
`sh:xone`

- **`sh:and`**: The focus node must satisfy all the shapes in the SHACL list.
- **`sh:or`**: The focus node must satisfy at least one of the shapes in the list.
- **`sh:xone`**: The focus node must satisfy exactly one of the shapes in the SHACL list.

Summary of Basic Constructs

Scope	Constraint Constructs
Classes and Data Types	class, datatype, nodeKind
Cardinality	minCount, maxCount
Values	node, in, hasValue
Ranges	minInclusive, maxInclusive, minExclusive, maxExclusive
Strings	minLength, maxLength, pattern, languageIn, uniqueLang
Property Pair Constraints	equals, disjoint, lessThan, lessThanOrEquals
Logical Operators	sh:not, sh:and, sh:or, sh:xone
Non-Validating Constraints	name, description, group, order, defaultValue
Qualified Shapes	qualifiedValueShape, qualifiedMinCount, qualifiedMaxCount
Assumptions	closed, ignoredProperties

Translation of a SHACL Constraint into SHACL-SPARQL

```
ex:InverseSSNConstraintShape
    a sh:NodeShape ;
    sh:targetObjectsOf ex:hasSSN ;
    sh:property [
        sh:path [ sh:inversePath ex:hasSSN ] ;
        sh:maxCount 1 ;
    ].
```

```
ex:InverseSSNConstraintShape a sh:NodeShape ;
    sh:targetSubjectsOf ex:hasSSN ;
    sh:sparql [
        a sh:SPARQLConstraint ;
        sh:message "The SSN must be unique." ;
        sh:select """
            SELECT $this
            WHERE {
                $this ex:hasSSN ?ssn .
                ?other ex:hasSSN ?ssn .
                FILTER ($this != ?other)
            }
        """ ;
    ] .
```

Note: To be valid, the RDF graph must not return any results when the SPARQL query is executed.

`$this represents the focus node.`

Overview of the SPARQL-based Validation Approach

Principle: SHACL-SPARQL constraints validate RDF graphs by ensuring that no constraint violation results are returned.

Steps:

1. Identify target nodes using `sh:targetClass`, `sh:targetSubjectsOf`, or `sh:targetObjectsOf`.
2. Define constraint conditions using a SPARQL `SELECT` query.
3. The query retrieves invalid nodes based on the constraint definition.
4. If the query returns results, the graph does not conform.

Powerful mechanism: Constraints bases on arbitrary graph patterns, aggregation queries can be expressed.

Initiatives for Discovering SHACL Shapes

Reference: Kashif Rabbani, Matteo Lissandrini, Katja Hose *Extraction of Validating Shapes from Very Large Knowledge Graphs*, VLDB 2023.

Type of Extracted SHACL Shapes

```
ex:ExtractionShape a sh:NodeShape ;
    sh:targetClass Tc ;
    sh:property [
        sh:path r ;
        sh:datatype Tp ; # or "sh:class Tp" if class constraint
        sh:minCount n ;
        sh:maxCount m ;
    ] .
```

Evaluation Metrics:

- **Support**: Number of entities that conform to a shape.
- **Confidence**: Ratio of conforming entities to total instances in `Tc`.

SHACL *vs* OWL

Can OWL keys be rewritten in SHACL?

Keys in OWL

OWL:HasKey: A mechanism to identify equivalent entities.

Example: Two authors of the same publication with the same name are considered the same person.

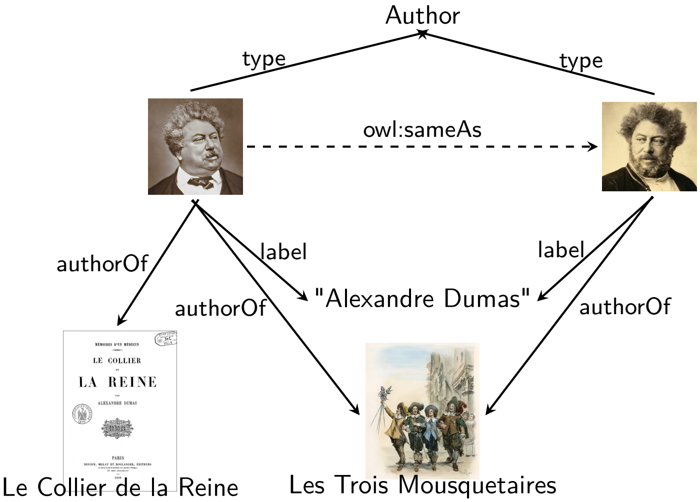
Expression in OWL 2:

```
owl:hasKey (Author label authorOf) .
```

Semantics: Two entities are **being considered identical** (and an `owl:sameAs` assertion **is being inferred**) if they *coincide* the specified properties.

The Term **coincide** in OWL

- OWL adopts the open world assumption (OWA).
- A fact cannot be refuted due to its absence in the knowledge base.
- Thus, **coincide** means *sharing at least one value in common*.



SHACL Rewriting

Key Idea: Enforcing uniqueness constraints in SHACL.

- SHACL is designed for validation of existing facts.
- New facts, such as `owl:sameAs`, cannot be inferred.
- Instead, SHACL can detect equivalent (duplicate) entities and raise validation errors.

SHACL-SPARQL Solution

Ensuring uniqueness of `(authorOf, label)`.

```
ex:AuthorKeyShape a sh:NodeShape ;
    sh:targetClass ex:Author ;
    sh:sparql [
        a sh:SPARQLConstraint ;
        sh:message "Duplicate (authorOf, label) combination
            detected.";
        sh:select """
            SELECT $this
            WHERE {
                $this ex:label ?label ;
                      ex:authorOf ?work .
                ?other ex:label ?label ;
                       ex:authorOf ?work .
                FILTER ($this != ?other)
            }
        """ ;
    ] .
```

Which other integrity constraints can SHACL express?

Unary Inclusion Dependencies

Example: Every person with a `schema:birthDate` must also have a `schema:birthPlace`.

Tuple-Generating Dependency (TGD) Representation:

$$\forall x, d \quad \texttt{birthDate}(x, d) \rightarrow \exists p \quad \texttt{birthPlace}(x, p)$$

SHACL Representation:

```
ex:BirthConstraintShape a sh:NodeShape ;
    sh:targetSubjectsOf schema:birthDate ;
    sh:property [
        sh:path schema:birthPlace ;
        sh:minCount 1 ;
    ].
```

Observation: This constraint is analogous to **foreign keys** in relational DBs.

Inclusion Dependencies (INDs)

Example: All parent relationships are also ancestor relationships.

TGD Representation:

$$\forall x, y \quad \texttt{hasParent}(x, y) \rightarrow \texttt{hasAncestor}(x, y)$$

SHACL Representation:

```
ex:ParentAncestorConstraint
    a sh:NodeShape ;
    sh:targetSubjectsOf ex:hasParent ;
    sh:property [
        sh:path ex:hasParent ;
        sh:equals ex:hasAncestor ;
    ] .
```

Conditional Inclusion Dependencies (CINDs)

Reference: CINDs (INDs + a filtering condition) for RDF were introduced in:
RDFind: Scalable Conditional Inclusion Dependency Discovery in RDF Datasets by Sebastian Kroll, Felix Naumann, et al., SIGMOD 2016.

Exemple: If a person is a graduate student, then they must have an undergraduate institution.

TGD Representation:

$$\forall x, y \quad (\texttt{rdf:type}(x, \texttt{gradStudent}) \rightarrow \exists y \quad \texttt{undergradFrom}(x, y))$$

SHACL Representation:

```
ex:GradStudentSubsetShape
    a sh:NodeShape ;
    sh:targetClass ex:GradStudent ;
    sh:property [
        sh:path ex:undergradFrom ;
        sh:minCount 1 ;
    ] .
```

Relation Functionality Constraint

Constraint: An entity must have at most one SSN (`ex:hasSSN`).

```
ex:SSNConstraintShape a sh:NodeShape ;
    sh:targetSubjectsOf ex:hasSSN ;
    sh:property [
        sh:path ex:hasSSN ;
        sh:maxCount 1 ;
    ].
```

Functional Relation: A relation r is said to be **functional** if:

$$\forall x, y_1, y_2, \quad r(x, y_1) \land r(x, y_2) \Rightarrow y_1 = y_2$$

Thus, each subject value is associated with at most one object value.

The concept of functionality is analogous to **functional dependencies**.

Functionality of the Inverse Relation

An individual's SSN is unique.

```
ex:InverseSSNConstraintShape a sh:NodeShape ;
    sh:targetObjectsOf ex:hasSSN ;
    sh:property [
        sh:path [ sh:inversePath ex:hasSSN ] ;
        sh:maxCount 1 ;
    ].
```

Conclusion: A property can be functional without its inverse being functional.

Functionality of the Inverse Relation

An individual's SSN is unique.

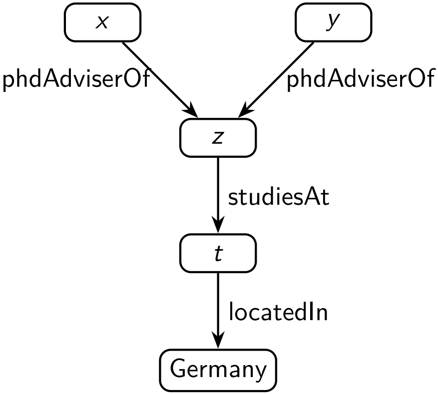
```
ex:InverseSSNConstraintShape a sh:NodeShape ;
    sh:targetObjectsOf ex:hasSSN ;
    sh:property [
        sh:path [ sh:inversePath ex:hasSSN ] ;
        sh:maxCount 1 ;
    ].
```

Conclusion: A property can be functional without its inverse being functional.

- OWL allows expressing the functionality of direct or inverse relations.
- SHACL extends this capability by allowing the functionality of paths.

Beyond Keys (or Conditional Keys)

Example: At a German university, two professors cannot supervise the same PhD student.



The constraint:
$\texttt{phdAdviserOf}(x, z)$,
$\texttt{phdAdviserOf}(y, z)$,
$\texttt{studiesAt}(z, t)$,
$\texttt{locatedIn}(t, '\texttt{Germany}') \Rightarrow x = y$

Questions:

- Why can't this key cannot be expressed in OWL?
- Can this constraint be expressed as a functional dependency?

Bibliography

- **W3C specification for SHACL**