Shape Expressions with Inheritance

Iovka Boneva[1][0000−0002−2696−7303], Jose Emilio Labra
Gayo[2][0000−0001−8907−5348], Eric Prud'hommeaux[3][0000−0003−1775−9921],
Katherine Thornton[4][0000−0002−4499−0451], and Andra
Waagmeester[5][0000−0001−9773−4008]

[1] Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
`iovka.boneva@univ-lille.fr` (corresponding author)
[2] WESO (WEb Semantics Oviedo) Research Group, University of Oviedo, Spain
`labra@uniovi.es`
[3] Janeiro Digital, USA `eric@uu3.org`
[4] Yale University Library, New Haven, CT, USA `katherine.thornton@yale.edu`
[5] Micelio BV, Ekeren, Belgium `andra@micelio.be`

Abstract. We formally introduce an inheritance mechanism for the
Shape Expressions language (ShEx). It is inspired by inheritance in
object-oriented programming languages, and provides similar advantages
such as reuse, modularity, and more flexible data modelling. Using an ex-
ample, we explain the main features of the inheritance mechanism. We
present its syntax and formal semantics. The semantics is an extension
of the semantics of ShEx 2.1. It also directly yields a validation algo-
rithm as an extension of the previous ShEx validation algorithms, while
maintaining the same algorithmic complexity.

Keywords: RDF · schemas for graphs · formal semantics of languages

1 Introduction

The Shape Expressions language (ShEx) was proposed in 2014 as a concise,
high-level language to describe and validate RDF data [12]. It allows us to define
ShEx schemas which are collections of shapes. A shape describes the required
properties of an RDF node and constrains their values. ShEx 2.1 was published
as a W3C final community report in 2019[6], and it introduced logical operators
on top of shapes, yielding *shape expressions*. They allow describing alternatives
with disjunction, or further constraining the shape of the data and data values
with conjunction. ShEx has been increasingly adopted in different domains such
as HL7 FHIR[7] and Wikidata [17], where it is used to describe data models and
to validate entity data.

The adoption of ShEx for describing large data models has led to the ap-
pearance of collaborative shape schema ecosystems like that of Wikidata[8]. An

---

[6] https://shex.io/shex-semantics-20191008/
[7] https://www.hl7.org/fhir/
[8] https://www.wikidata.org/wiki/Wikidata:Database_reports/EntitySchema_directory

important requirement for such ecosystems is to support reusability. The simplest feature for reuse is to import a shape expression that has been defined elsewhere. ShEx 2.1 does offer this possibility and the imported shape expression can be reused as is. Data modelling and reuse often require adapting the existing data and its model to one's particular needs. Two common adaptation patterns are to *restrict* the allowed values of existing properties, and to *extend* the data model and the data with properties that are specific to a new application. Additionally, a commonly-used feature is to define frequently occurring patterns which serve as building blocks of shape expressions.

In response to submitted use cases, the ShEx Community Group[9] has proposed an inheritance mechanism for ShEx that provides the afore-mentioned features. Inspired by inheritance in programming languages, it allows for the definition of a hierarchy of shape expressions. Child shapes can extend their parents with new required properties for RDF nodes, and can impose additional constraints on the properties of their parents. Additionally, as in object-oriented programming languages, a node with a child shape can be used where a parent shape is expected. Multiple inheritance is also allowed. Finally, shape expressions can be made *abstract* indicating that they cannot be used on their own, but only as building blocks of other expressions. An early version of the inheritance mechanism has been used in [13] to translate in ShEx the user-facing documentation of data structures in FHIR.

In this paper, we introduce the formal semantics for ShEx with inheritance, built as an extension of the formal semantics of ShEx 2.1 [4]. The formalization posed some challenges, notably for handling multiple inheritance. In particular, we introduce a syntactic restriction on the combined use of inheritance with disjunction. The restriction avoids unnecessary complexity in the language, while still providing the functionality required by the use cases. The ShEx formalization presented here will be submitted for the next version of the ShEx language, currently under standardization by IEEE.[10]

The paper is organized as follows. In Sect. 2 we give an example illustrating the main features of the inheritance mechanism, and define some preliminary notions. Sect. 3 gives the syntax, and Sect. 4 the semantics of the language. Sect. 5 presents a validation algorithm. We close with a discussion, an overview of related work and a conclusion in Sect. 6. The proofs omitted in the paper are presented in appendix. A companion webpage [1] lists the current implementations and provides source code and demonstrations for the examples.

2   Motivating example and background

Assume the following constraints on nodes in an RDF graph: T_{str} is satisfied by string literals, T_{float} by float literals, T_{any} by all RDF nodes, $T_{"colour"}$ is satisfied by the literal "colour", and $T_{"radius"}$ by the literal "radius". A ShEx schema using inheritance is presented in Fig. 1, we will describe it section by section.

---

[9] https://www.w3.org/community/shex/
[10] https://standards.ieee.org/ieee/3330/11119/

$$Coord \rightarrow \langle\, x\, @T_{float}\ ;\ y\, @T_{float}\,\rangle$$
$$Attribute \rightarrow \mathsf{extends}\ \_\ \langle\, name\, @T_{str}\ ;\ value\, @T_{any}\,\rangle$$
$$Colour \rightarrow \mathsf{extends}\ Attribute\ \langle\, scope\, @T_{str}\,\rangle\ \ \mathsf{and}\ \ \langle\, name\, @T_{\text{"colour"}}\,\rangle$$

$$\mathsf{abstract}\ Figure \rightarrow \mathsf{extends}\ \_\ \langle\, coord\, @Coord\,\rangle$$
$$Circle \rightarrow \mathsf{extends}\ Figure\ \langle\, attr\, @Radius\,\rangle$$
$$Radius \rightarrow \mathsf{extends}\ Attribute\ \langle\, \varepsilon\,\rangle\ \mathsf{and}\ \langle\, name\, @T_{\text{"radius"}}\ ;\ value\, @T_{float}\,\rangle$$

$$ColouredFigure \rightarrow \mathsf{extends}\ Figure\ \langle\, attr\, @Colour\,\rangle$$
$$ColouredCircle \rightarrow \mathsf{extends}\ Circle,\ ColouredFigure\ \langle\, \varepsilon\,\rangle$$

Fig. 1. A ShEx schema with inheritance.

The source code for that schema is available on the companion webpage [1]. The shape expression named *Coord* describes nodes representing coordinates, i.e. having properties x and y whose values are floats. The shape expression *Attribute* requires a property *name* which is a string, and a property *value* that can be anything. The extends _ indicates that the shape expression is being extended. A *Colour* is a specific kind of attribute (extends on *Attribute*) that additionally to the *name* and *value* properties requires a *scope* property whose value is a string. This is captured by the first part of the definition, preceding the and keyword. The second part states that the *name* of a *Colour* attribute is "colour". Thus, inheritance provides a mechanism for requiring additional properties (before the and) and restricting existing properties (after the and). Fig. 2 shows an example graph. Nodes a1, a2, a3 satisfy *Attribute*, node a2 satisfies *Colour*, and node c1 satisfies *Coord*.

A *Figure* has coordinates associated with the property *coord*. The shape expression *Figure* is marked as abstract, which indicates that it cannot be directly satisfied, rather one of its non-abstract descendants will be satisfied. A *Circle* is a *Figure* that has a *Radius* attribute reachable through the property *attr*, which in turn is an *Attribute* having *name* "radius" and a float *value*. Note that the definition of *Circle* does not have a restriction after the and; in fact the restriction is optional. On the other hand, the extending portion of *Radius* is ε, meaning that a *Radius* does not require additional properties beyond those specified in its parent *Attribute*. In the example graph, f2 is a *Circle*, a1, a3 satisfy *Radius*.
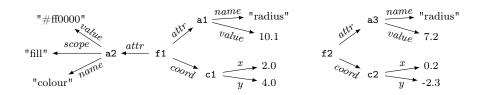

Fig. 2. Example graph with nodes f1, f2, a1, a2, a3 and strings and float literals.
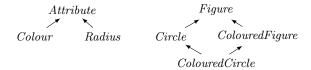
Fig. 3. Extension hierarchy graph for the example.

The final part of the example illustrates multiple inheritance. A *ColouredCircle* extends both on *ColouredFigure* and *Circle* which in turn both extend from *Figure*. Recall that a *Figure* has one *coord* property, which is inherited twice by *ColouredCircle*. The inheritance mechanism ensures that multiply-inherited properties are not used more than once. In this particular example, a *Coloured-Circle* can have only one *coord*. In the example graph, `f1` is a *ColouredCircle*.

As in object-oriented programming languages, inheritance yields an extension hierarchy which is a directed acyclic graph. Fig. 3 depicts the inheritance graph of the example schema, with arrows going from shape expressions to their direct ancestors. The inheritance mechanism requires that if a node satisfies some of the descendants of a shape expression, then it satisfies the shape expression itself. Therefore, `f1` is a *ColouredCircle*, a *ColouredFigure*, a *Circle*, and a *Figure*.

Background IRI, Blank and Literal are three countable mutually disjoint sets of IRIs, blank nodes, and literals, respectively. Elements of $(\text{IRI} \cup \text{Blank}) \times \text{IRI} \times (\text{IRI} \cup \text{Literal} \cup \text{Blank})$ are called triples. If (n, p, o) is a triple, then n is its subject, p its property, and o its object. An RDF graph $\mathbf{G}$ is a finite set of triples. The set of nodes of $\mathbf{G}$ is the set of all subjects and objects of its triples and is denoted $nodes(\mathbf{G})$. Given a graph $\mathbf{G}$ and a node $n \in nodes(\mathbf{G})$, the *neighbourhood* of n in $\mathbf{G}$ is the set $neigh_{\mathbf{G}}(n)$ of triples having n as subject. Formally, $neigh_{\mathbf{G}}(n) = \{(n, p, o) \mid (n, p, o) \in \mathbf{G}\}$. A set of triples $M \subseteq \text{Triples}$ is called a *neighbourhood set of triples* if all its elements have the same subject denoted $subject(M)$.

We use the symbol $\uplus$ for disjoint union, i.e. $M = M_1 \uplus \cdots \uplus M_k$ (for some $k \in \mathbb{N}$) means that $M = M_1 \cup \cdots \cup M_k$ and $M_i \cap M_j = \emptyset$ whenever $i \neq j$. The disjoint union of zero sets is the empty set, i.e., if $k = 0$, then $M = \emptyset$. In grammars or expressions, square brackets [ ] surround optional elements.

3   Syntax

A ShEx schema consists of a set of labeled (named) shape expressions to be checked for conformance with the nodes of an RDF graph. We start by defining shape expressions, then give a formal definition of ShEx schemas with inheritance and the corresponding extension hierarchy.

Definition 1 (shape expressions). *Let* $\mathbf{Y}$ *be a finite set of* simple labels *and* $\mathbf{X}$ *be a finite set of* extendable labels *disjoint from* $\mathbf{Y}$*. A shape expression* over

$\mathbf{Y}$ *and* $\mathbf{X}$ *is either a* reference *of the form @z, or a* plain shape expression *derivable from the non-terminal u in the following grammar:*

$u ::= h \mid t \mid c \mid u \text{ or } u \mid u \text{ and } u \mid \text{ not } u$. *(plain shape expression)*

$h ::= [\text{closed}] \; [\text{extra } P] \; \langle e \rangle$. *(shape)*

$t ::= \text{extends } X \; h$. *(shape with extends)*

$c ::= a \; boolean \; function \; over \; \mathsf{IRI} \cup \mathsf{Blank} \cup \mathsf{Literal}$. *(node constraint)*

$e ::= \varepsilon \mid p \, @z \mid e \mid e \mid e \, ; e \mid e*$. *(triple expression)*

where $z \in \mathbf{Y} \cup \mathbf{X}$, $X \subseteq \mathbf{X}$, $p \in \mathsf{IRI}$, $P \subseteq \mathsf{IRI}$, *the symbol 'ε' is a constant, and the symbols '@', '$|$', '$*$' and ';' are operators. An expression derivable by the non-terminal h, resp. t, resp. c, resp. e, is called a* shape, *resp. a* shape with extends, *resp. a* node constraint, *resp. a* triple expression*. An expression of the form p @z is called a* triple constraint*.

A shape with extends is a new kind of shape expression. It is constructed from a (possibly empty) set X of extendable labels and a shape h. The labels in X indicate which shape expressions are being extended. All other ingredients of shape expressions are present in ShEx 2.1.

Definition 2 (ShEx schema). *A* ShEx schema (with inheritance) *is a tuple* $\mathbf{S} = (\mathbf{Y}, \mathbf{X}, \mathbf{def}, \mathbf{A})$, *where* $\mathbf{Y}$ *is a finite set of* simple labels, $\mathbf{X}$ *is a finite set of* extendable labels *disjoint from* $\mathbf{Y}$, $\mathbf{A} \subseteq \mathbf{X}$ *are the* abstract *labels, and* $\mathbf{def}$ *is a function in which every label from* $\mathbf{Y} \cup \mathbf{X}$ *associates its definition, with:*

– *if* $y \in \mathbf{Y}$*, then* $\mathbf{def}(y)$ *is a plain shape expression;*
– *if* $x \in \mathbf{X}$*, then* $\mathbf{def}(x)$ *is a shape expression of the form* extends $X \; h \; [\text{and } u]$, *also called an* extendable shape expression, *where h is a shape and u is a plain shape expression.*

Example 1. In Sect. 2 we presented a schema – denote it $\mathbf{S}_1 = (\mathbf{Y}_1, \mathbf{X}_1, \mathbf{def}_1, \mathbf{A}_1)$ – as a set of rules of the form $z \to s$, to be understood as $\mathbf{def}_1(z) = s$. The simple labels are $\mathbf{Y}_1 = \{Coord, T_{any}, T_{str}, T_{float}, T_{\text{"colour"}}, T_{\text{"radius"}}\}$. The definitions of the sub-scripted T's correspond to the node constraints described in the text, e.g., $\mathbf{def}_1(T_{str})$ is the function that is true for string literals and false for all other nodes. The extendable labels are those that appear on the left-hand sides of the rules, except for $Coord$, i.e. $\mathbf{X}_1 = \{Attribute, Colour, \ldots, ColouredCircle\}$. Their definitions are extendable shape expressions, in which we omit the curly braces around the elements of the set X and write _ if it is empty. Finally, $\mathbf{A}_1 = \{Figure\}$ is indicated by the `abstract` keyword.

In the sequel, all examples of schemas will use the same notation as the one used in Sect. 2 and explained in Example 1.

Definition 3 (extension hierarchy). *The* extension hierarchy *of the schema* $\mathbf{S} = (\mathbf{Y}, \mathbf{X}, \mathbf{def}, \mathbf{A})$ *is a directed graph denoted* $H_\mathbf{S}$ *whose set of nodes is* $\mathbf{X}$ *and that has an edge from x to x' if and only if* $\mathbf{def}(x) = $ extends $X \; h \; [\text{and } u]$ *and*

$x' \in X$. If there exists a (possibly empty) path from x to x' in $H_{\mathbf{S}}$, we say that x' is an ancestor of x or, equivalently, that x is a descendant of x'. We denote $anc(x)$, resp. $desc(x)$, the set of ancestors, resp. descendants, of the label x.

Example 2. Fig. 3 shows $H_{\mathbf{S}_1}$, the extension hierarchy of the schema $\mathbf{S}_1$ from Example 1. Note that the extension hierarchy is determined only by the first shape with extends in the definition of an extendable shape label. For instance, let $\mathbf{def}(x) = $ extends $x_1, x_2 \langle \varepsilon \rangle$ and extends $x_3, x_4 \langle \varepsilon \rangle$. According to Def. 3, x_1 and x_2 are ancestors of x, but x_3 and x_4 are not ancestors of x.

4   Semantics

Let $\mathbf{G}$ be a graph and $\mathbf{S} = (\mathbf{Y}, \mathbf{X}, \mathbf{def}, \mathbf{A})$ be a ShEx schema. A *typing of $\mathbf{G}$ by $\mathbf{S}$* is a subset of $nodes(\mathbf{G}) \times (\mathbf{Y} \cup \mathbf{X})$. We use typings to define the semantics of triple expressions and shape expressions. Intuitively, a typing can be understood as a set of assertions about which nodes of the graph satisfy which shape expressions. It can be used to derive other such assertions. In Sect. 4.1 we define the semantics of expressions under a given typing. A typing τ is *correct* (w.r.t. a graph and a schema) if each of the assertions it contains can be derived from τ itself. In Sect. 4.2 we introduce well-defined schemas by forbidding some circular dependencies between shape expressions. Finally, in Sect. 4.3 we show that if a schema is well-defined, then there exists a unique maximal correct typing $\tau_{\max}$ which defines the semantics of ShEx schemas, in the sense that it contains all correct assertions about which nodes satisfy which shape expressions.

4.1   Satisfying an expression under given typing

The semantics of shape expressions and triple expressions are given by the means of three mutually recursive satisfiability relations. For triple expressions, the satisfiability relation is of the form $\mathbf{G}, M, \tau \models_{\mathbf{S}} e$ and for shape expressions we use two satisfiability relations, one of the form $\mathbf{G}, n, \tau \models_{\mathbf{S}} s$ and another one of the form $\mathbf{G}, M, \tau \models_{\mathbf{S}} s$ (where $\mathbf{S}$ is a schema, $\mathbf{G}$ is a graph, n is a node of $\mathbf{G}$, τ is a typing over $\mathbf{G}$ and $\mathbf{S}$, M is a set of neighbourhood triples from $\mathbf{G}$, e is a triple expression, and s is a shape expression). The different signatures of the three relations will always allow distinguishing them. We will omit $\mathbf{G}$ and $\mathbf{S}$ whenever they are clear from the context. The satisfiability relation of the form $\mathbf{G}, M, \tau \models_{\mathbf{S}} s$ is new for ShEx with inheritance. It is necessary for the definition of the semantics of shapes with extends.

For the remainder of the section, consider given a graph $\mathbf{G}$ and a ShEx schema $\mathbf{S} = (\mathbf{Y}, \mathbf{X}, \mathbf{def}, \mathbf{A})$ that will be implicit (omitted) in all satisfiability relation statements. The semantics of the satisfiability relations are given in Tables 1, 2 and 3 inductively on the structure of expressions. The right-hand side column in each table is a boolean expression giving the truth value for the case from the left-hand side column. The symbols $\neg, \wedge, \vee, \exists$ and $\implies$ have their usual meaning from logic. The symbol $=$ is test of equality, while $:=$ is assignment.

Table 1. Definition of the satisfiability relation $M, \tau \models e$

	e	$M, \tau \models e$
1.	ε	$M = \emptyset$
2.	$p\ @z$	$M = \{(n, p, o)\} \quad \wedge \quad o, \tau \models @z$
3.	$e_1 \mid e_2$	$M, \tau \models e_1 \quad \vee \quad M, \tau \models e_2$
4.	$e_1\ ;\ e_2$	$\exists M_1, M_2.\ M = M_1 \uplus M_2 \quad \wedge \quad M_1, \tau \models e_1 \quad \wedge \quad M_2, \tau \models e_2$
5.	$e_1 *$	$\exists k \in 0..|M|.\ \exists M_1, \ldots, M_k.$ $M = M_1 \uplus \cdots \uplus M_k \quad \wedge \quad \forall i \in 1..k.\ M_i, \tau \models e_1$

Table 2. Definition of the satisfiability relation $n, \tau \models s$.

	s	$n, \tau \models s$
6.	c	$c(n)$
7.	s_1 or s_2	$n, \tau \models s_1 \quad \vee \quad n, \tau \models s_2$
8.	s_1 and s_2	$n, \tau \models s_1 \quad \wedge \quad n, \tau \models s_2$
9.	not s_1	$n, \tau \neg \models s_1$
10.	$@z$	$(n, z) \in \tau$
11.	$[\,\mathsf{closed}\,]\ [\,\mathsf{extra}\ P\,]\ \langle\, e\, \rangle$	$neigh_{\mathbf{G}}(n), \tau \models [\,\mathsf{closed}\,]\ [\,\mathsf{extra}\ P\,]\ \langle\, e\, \rangle$
12.	$\mathsf{extends}\ X\ h$	$neigh_{\mathbf{G}}(n), \tau \models \mathsf{extends}\ X\ h$

When $\mathbf{G}, M, \tau \models_{\mathbf{S}} e$, resp. $\mathbf{G}, n, \tau \models_{\mathbf{S}} s$, resp. $\mathbf{G}, M, \tau \models_{\mathbf{S}} s$ holds, we say that M *satisfies* e, resp. n *satisfies* s, resp. M *satisfies* s under typing τ. In the sequel of this section we explain the satisfiability relations.

Triple expressions (Table 1) The semantics of triple expressions are the same as those of ShEx 2.1. Only the empty set of triples satisfies the empty expression ε. A triple constraint $p\ @z$ is satisfied by a singleton set which unique triple has p as property and an object that satisfies $@z$. The expression $e_1 \mid e_2$ is satisfied if one of e_1 or e_2 is satisfied. As for $e_1\ ;\ e_2$, it is satisfied by M if M can be split in two parts M_1 and M_2 that satisfy the sub-expressions e_1 and e_2, respectively. Finally, the repeated expression $e*$ is satisfied by M if M can be split into k parts, for some k less than or equal to the number of elements in M, and each of them satisfies e. Note that the empty set always satisfies $e*$.

Example 3. Let τ be the typing and $M_{24}, M_{246}, M_{2a}, M_{24a}$ be the sets of triples defined below. Consider the triple expression $e = p\ @T_{even}\ ;\ (p\ @T_{<5}{}^* \mid p\ @T_{str})$. Then $M_{24}, \tau \models e$, $M_{246}, \tau \models e$, $M_{2a}, \tau \models e$, but $M_{24a}, \tau \neg \models e$.

$$M_{24} = \{(n, p, 2), (n, p, 4)\} \qquad M_{246} = \{(n, p, 2), (n, p, 4), (n, p, 6)\}$$
$$M_{2a} = \{(n, p, 2), (n, p, \texttt{"a"})\} \qquad M_{24a} = \{(n, p, 2), (n, p, 4), (n, p, \texttt{"a"})\}$$
$$\tau = \{(2, T_{even}), (2, T_{<5}), (4, T_{even}), (4, T_{<5}), (6, T_{even}), (6, T_{>5}), (\texttt{"a"}, T_{str})\}$$

Shape expression satisfied by a node (Table 2) A node constraint c is satisfied by n if $c(n)$ holds. The semantics of the boolean operators or, and and not are

Table 3. Definition of the satisfiability relation $M, \tau \models s$.

	s	$M, \tau \models s$
13.	c	$c(subject(M))$
14.	s_1 or s_2	$M, \tau \models s_1 \quad \vee \quad M, \tau \models s_2$
15.	s_1 and s_2	$M, \tau \models s_1 \quad \wedge \quad M, \tau \models s_2$
16.	not s_1	$M, \tau \neg \models s_1$
17.	$[\,$closed$\,]\,[\,$extra $P\,]\langle\, e\,\rangle$	$M^e, \tau \models e \quad \wedge \quad props(M^{\notin}) \subseteq P' \quad \wedge$ closed is present $\implies props(M) \subseteq props(e) \cup P'$ with $\quad P' := P$ if extra is present, $P' := \emptyset$ otherwise $\quad M^e := \{(n, p, o) \in M \mid \exists p\, @z \in tcs(e).\, o, \tau \models @z\}$ $\quad M^{\notin} := \{(n, p, o) \in M \mid p \in props(e)\} \setminus M^e$
18.	extends $X\ h$	$\exists M'$ and $\exists M_x$ for every $x \in anc(X)$ such that $\quad M = M' \uplus \biguplus_{x \in anc(X)} M_x \quad \wedge$ $\quad M', \tau \models h \quad \wedge$ $\quad$ for every $x \in anc(X).$ $\qquad M_x, \tau \models ext\text{-}te(x) \quad \wedge$ $\qquad restr(x)$ present $\implies \left(\bigcup_{z \in anc(x)} M_z \right), \tau \models restr(x)$

as expected. A reference $@z$ is satisfied by node n if the typing contains the pair (n, z). Both a shape and a shape with extends are satisfied by a node if the node's neighbourhood satisfies the shape.

Shape expression satisfied by a set of triples (Table 3). A node constraint is satisfied by a set M of neighbourhood triples if their common subject satisfies the constraint. The definitions of boolean operators are as expected. For a shape (line 17.), we use the following notations: $props(M) = \{p \mid (n, p, o) \in M\}$ is the set of properties of the triples in M, $tcs(e)$ is the set of triple constraints sub-expressions of the triple expression e, and $props(e) = \{p \mid p\, @s \in tcs(e)\}$ is the set of properties in the triple constraints of e. We identify two significant subsets of M:

- M^e are the triples from M that satisfy some triple constraint in e under τ,
- $M^{\notin}$ are the triples from M whose property appears in e but satisfy none of the triple constraints in e.

We require that M^e satisfies the triple expression e, while the triples in $M^{\notin}$ may use only extra properties from P. In particular, $M^{\notin}$ must be empty if extra P is not present. Additionally, if the shape is closed, then M contains only triples whose properties appear in e or are in the set P of extra properties.

Example 4. Let τ and $M_{24}, M_{246}, M_{2a}, M_{24a}$ be as in Example 3 and define the triple expression $e = p\, @T_{even}\, ;\, p\, @T_{<5}$. Then, for the sets used in line 17. we

have e.g.: $M_{24a}^e = \{(n_2, p, 2), (n_2, p, 4)\}$, $M_{24a}^{\notin} = \{(n_2, p, \texttt{"a"})\}$, $M_{246}^e = M_{246}$ and $M_{246}^{\notin} = \emptyset$. For example, the following hold:

$$M_{24}, \tau \models \langle\, e\,\rangle \qquad M_{24a}, \tau \models \textsf{extra}\,\{p\}\,\langle\, e\,\rangle \qquad\qquad M_{246}, \tau \,\neg\models \textsf{extra}\,\{p\}\,\langle\, e\,\rangle$$
$$M_{2a}, \tau \,\neg\models \langle\, e\,\rangle \quad M_{24a} \cup \{(n, q, 2)\}, \tau \models \textsf{extra}\,\{p\}\,\langle\, e\,\rangle$$
$$M_{246}, \tau \,\neg\models \langle\, e\,\rangle \quad M_{24a} \cup \{(n, q, 2)\}, \tau \,\neg\models \textsf{closed}\ \textsf{extra}\,\{p\}\,\langle\, e\,\rangle$$

For the semantics of shapes with extends (line 18.), we use the following notations. If $X \in \mathbf{X}$, then $anc(X) = \bigcup_{x \in X} anc(x)$ is the union of the ancestor sets of the labels in X; remark that $X \subseteq anc(X)$. For an extendable label $x \in \mathbf{X}$ and assuming $\mathbf{def}(x) = \textsf{extends}\,X\ h\ [\ \textsf{and}\ \ s\,]$, we denote by $ext\text{-}te(x)$ the triple expression of h, and by $restr(x)$ the shape expression s whenever it exists.

For a shape with extends $t = \textsf{extends}\,X\ h$, different parts of M will be used to satisfy h and the shape expressions extended in t. First, M must be partitioned as $M' \uplus M_{x_1} \uplus \cdots \uplus M_{x_k}$ where the x_i are the strict ancestors of t, i.e. $\{x_1, \ldots, x_k\} = anc(X)$ (for some $k \in \mathbb{N}$). Assume also that $\mathbf{def}(x_i) = \textsf{extends}\,X_i\ h_i\ [\ \textsf{and}\ \ s_i\,]$ for every $i \in 1..k$. Then we require that:

- M' satisfies h and for every $i \in 1..k$, M_{x_i} satisfies the triple expression $ext\text{-}te(x_i)$ which is the triple expression of the shape h_i;
- if present, the restriction s_i must be satisfied not by M_{x_i} alone, but by the union of all the M_z such that z is ancestor of x_i (recall that $x_i \in anc(x_i)$).

Example 5. Let $\mathbf{Y} = \{T_{even}, T_{<5}, T_{>5}\}$ and let $\tau, M_{24}, M_{2a}, M_{246}, M_{24a}$ be as in Example 3. Consider the schema $\mathbf{S} = (\mathbf{Y}, \{x_1, \ldots, x_6\}, \mathbf{def}, \emptyset)$, where $\mathbf{def}$ is given by:

$x_0 \to \textsf{extends}\,\_\,\langle\, p\ @T_{even}\,\rangle$
$x_1 \to \textsf{extends}\,x_0\,\langle\, p\ @T_{even}\,\rangle$ $\qquad\qquad\qquad$ $x_2 \to \textsf{extends}\,x_1\,\langle\, p\ @T_{<5}\,\rangle$
$x_3 \to \textsf{extends}\,x_0\,\langle\, p\ @T_{even}\,\rangle\ \textsf{and}\ \langle\, p\ @T_{>5}{}^*\,\rangle$ $\quad$ $x_4 \to \textsf{extends}\,x_6\,\langle\, p\ @T_{<5}\,\rangle$
$x_5 \to \textsf{extends}\,x_0\ \textsf{extra}\,\{p\}\,\langle\, \varepsilon\,\rangle$ $\qquad\qquad\quad$ $x_6 \to \textsf{extends}\,x_3\,\langle\, p\ @T_{even}\,\rangle$

That is, $\mathbf{def}(x_1)$ requires exactly two p-triples with even values, and $\mathbf{def}(x_2)$ requires an additional p-triple which value is less than five. Then, e.g., $M_{24}, \tau \models \mathbf{def}(x_1)$ and $M_{246}, \tau \models \mathbf{def}(x_2)$, but $M_{24}, \tau\neg \models \mathbf{def}(x_2)$ and $M_{24a}, \tau\neg \models \mathbf{def}(x_1)$. In comparison, $\mathbf{def}(x_3)$ requires exactly two p-triples with even values greater than five, the latter being expressed by the restriction after the and. Therefore, none of the sets from Example 3 satisfies neither $\mathbf{def}(x_3)$, nor $\mathbf{def}(x_4)$, under τ. Finally, $M_{2a}, \tau \models \mathbf{def}(x_5)$ because $\mathbf{def}(x_5)$ allows the extra triple $(n, p, \texttt{"a"})$. However, $M_{24a}, \tau\neg \models\mathbf{def}(x_6)$, because $\textsf{extra}\{p\}$ in the extended shape expression $\mathbf{def}(x_5)$ is not considered for satisfying $\mathbf{def}(x_6)$, thus nothing allows the triple $(n, p, \texttt{"a"})$.

We will be interested in correct typings only. They are coherent w.r.t. the satisfiability relation, and they ensure that an abstract label can be satisfied only through one of its non-abstract descendants.

Definition 4 (correct typing). *Let* $\mathbf{G}$ *be a graph and* $\mathbf{S}$ *be a schema. A typing* τ *of* $\mathbf{G}$ *by* $\mathbf{S}$ *is* correct *if for every* $(n, z) \in \tau$:

- *if* $z \notin \mathbf{A}$, *then* $\mathbf{G}, n, \tau \models_{\mathbf{S}} \mathbf{def}(z)$,
- *if* $z \in \mathbf{A}$, *then there exists* $x \in desc(z) \setminus \mathbf{A}$ *such that* $\mathbf{G}, n, \tau \models_{\mathbf{S}} \mathbf{def}(x)$.

We denote $\mathcal{C}(\mathbf{G}, \mathbf{S})$ *the set of correct typings of* $\mathbf{G}$ *by* $\mathbf{S}$.

4.2   Well-defined schemas

Shape expressions can refer to each other or extend one another and this could result in circular definitions. Well-defined schemas forbid some circular definitions. We start by defining a dependency graph that captures how shape expressions depend on one another, then use it to define well-defined schemas.

For every shape expression or triple expression s, let $sub\text{-}expr(s)$ be the set of its sub-expressions, viewed as syntactic objects. Additionally, $neg\text{-}sub\text{-}expr(s)$ is the set of sub-expressions of s that appear in s under an odd number of occurrences of the negation operator not. Let $\mathbf{S} = (\mathbf{Y}, \mathbf{X}, \mathbf{def}, \mathbf{A})$ be a schema. We define the following four binary relations for labels $z, z' \in \mathbf{X} \cup \mathbf{Y}$:

- $dep\text{-}extends_{\mathbf{S}}(z, z')$ iff $H_{\mathbf{S}}$ has an edge from z to z' or from z' to z;
- $dep_{\mathbf{S}}(z, z')$ iff z' appears as a reference in $\mathbf{def}(z)$, i.e. $@z' \in sub\text{-}expr(\mathbf{def}(z))$;
- $dep\text{-}shape\text{-}neg_{\mathbf{S}}(z, z')$ iff $[\text{closed}][\text{extra } P]\langle\, e \,\rangle \in neg\text{-}sub\text{-}expr(\mathbf{def}(z))$ and $@z' \in sub\text{-}expr(e)$;
- $dep\text{-}extra\text{-}neg_{\mathbf{S}}(z, z')$ iff $[\text{closed}][\text{extra } P]\langle\, e \,\rangle \in sub\text{-}expr(\mathbf{def}(z))$, and $p\; @z' \in tcs(e)$, and $p \in P$.

The dependencies *dep-shape-neg* and *dep-extra-neg* are called *negative* dependencies. Obviously, *dep-extends* is new for ShEx with inheritance. Observe that *dep* subsumes *dep-shape-neg* and *dep-extra-neg*. We say that z depends on z' in schema $\mathbf{S}$ if $dep_{\mathbf{S}}(z, z')$ or $dep\text{-}extends_{\mathbf{S}}(z, z')$. The schema subscript is omitted whenever the schema is clear from the context.

Example 6. Let $\mathbf{S}_1$, $\mathbf{S}_2$ and $\mathbf{S}_3$ be the schemas below, from left to right, with respective sets of simple labels $\mathbf{Y}_1 = \{y_1, y_2, y_3\}$, $\mathbf{Y}_2 = \{y_4, y_5, y_6\}$ and $\mathbf{Y}_3 = \{y_7, y_8\}$ and with extendable labels $\mathbf{X}_3 = \{x_1, x_2\}$ for $\mathbf{S}_3$. Below each schema are enumerated the facts of its dependency relations.

$y_1 \to \langle\, p\; @y_2 \,\rangle$ and $\langle\, q\; @y_3 \,\rangle$	$y_4 \to \langle\, p\; @y_5 \,\rangle$ or $\langle\, q\; @y_6 \,\rangle$	$x_1 \to \text{extends } \_ \; \langle\, p\; @y_7 \,\rangle$
$y_2 \to \langle\, q\; @y_1 \,\rangle$	$y_5 \to \text{not}\langle\, q\; @y_4 \,\rangle$	$x_2 \to \text{extends } x_1\; \langle\, p\; @y_8 \,\rangle$
$y_3 \to \text{extra}\{r\}\; \langle\, p\; @y_1 \,\rangle$	$y_6 \to \text{extra}\{p\}\; \langle\, p\; @y_4 \,\rangle$	$y_7 \to \text{not}\; \langle\, q\; @x_2 \,\rangle$
		$y_8 \to c$
$dep_{\mathbf{S}_1}(y_1, y_2) \quad dep_{\mathbf{S}_1}(y_1, y_3)$	$dep_{\mathbf{S}_2}(y_4, y_5)$	$dep_{\mathbf{S}_3}(x_1, y_7)\; dep_{\mathbf{S}_3}(x_2, y_8)$
$dep_{\mathbf{S}_1}(y_2, y_1) \quad dep_{\mathbf{S}_1}(y_3, y_1)$	$dep_{\mathbf{S}_2}(y_4, y_6)$	$dep\text{-}shape\text{-}neg_{\mathbf{S}_3}(y_7, x_2)$
	$dep\text{-}shape\text{-}neg_{\mathbf{S}_2}(y_5, y_4)$	$dep\text{-}extends_{\mathbf{S}_3}(x_1, x_2)$
	$dep\text{-}extra\text{-}neg_{\mathbf{S}_2}(y_6, y_4)$	$dep\text{-}extends_{\mathbf{S}_3}(x_2, x_1)$

The dependency graph of a schema $\mathbf{S}$ regroups all these dependency relations and, together with the extension hierarchy, is used to define well-defined schemas.

Definition 5 (dependency graph). *Let* $\mathbf{S} = (\mathbf{Y}, \mathbf{X}, \mathbf{def}, \mathbf{A})$ *be a schema. The* dependency graph *of* $\mathbf{S}$ *is a directed labelled graph denoted* $D_{\mathbf{S}}$ *whose set of vertices is* $\mathbf{Y} \cup \mathbf{X}$ *and that has an edge from* z *to* z' *labelled* d *if* $d(z, z')$ *holds, where* d *is one of the dependency relations* dep-extends$_{\mathbf{S}}$, dep$_{\mathbf{S}}$, dep-shape-neg$_{\mathbf{S}}$, dep-extra-neg$_{\mathbf{S}}$.

Definition 6 (well-defined schema, stratified negation). *A schema* $\mathbf{S}$ *is* well-defined *if it satisfies these conditions:*

 - $H_{\mathbf{S}}$, *the extension hierarchy graph of a* $\mathbf{S}$, *is acyclic;*
 - *in* $D_{\mathbf{S}}$, *no cycle contains a negative dependency edge labelled* dep-shape-neg *or* dep-extra-neg. *We say in this case that* $\mathbf{S}$ *is* with stratified negation.

Example 7. Considering the schemas from Example 6: $\mathbf{S}_1$ is well-defined, as the cycles along y_1, y_2 and along y_1, y_3 are not forbidden; $\mathbf{S}_2$ is not well-defined as its dependency graph contains two cycles – one along y_4, y_5, and another one along y_4, y_6 – each containing a negative dependency; $\mathbf{S}_3$ is not well-defined as it has a forbidden cycle *dep-shape-neg*$_{\mathbf{S}_3}(y_7, x_2)$, *dep-extends*$_{\mathbf{S}_3}(x_2, x_1)$, *dep*$_{\mathbf{S}_3}(x_1, y_7)$. Note that the cycle won't be present without the *dep-extends* relation; this illustrates that extension and negation cannot be arbitrarily interleaved.

From here we will discuss only well-defined schemas. For well-defined schemas, satisfiability relations can be effectively computed.

Lemma 1. *The evaluation of* $\mathbf{G}, n, \tau \models_{\mathbf{S}} \mathbf{def}(z)$ *always terminates, for every schema* $\mathbf{S} = (\mathbf{Y}, \mathbf{X}, \mathbf{def}, \mathbf{A})$, *graph* $\mathbf{G}$, *node* n *of* $\mathbf{G}$, *typing* τ *of* $\mathbf{G}$ *by* $\mathbf{S}$, *and label* $z \in \mathbf{Y} \cup \mathbf{X}$.

4.3   Semantics of ShEx Schemas

The remaining of the section is devoted to the construction of the maximal correct typing. It is defined w.r.t. a stratification of the schema, which defines stratums (layers) of labels such that circular dependencies can happen only between labels on the same stratum, while negative dependencies can only go from upper to lower stratums.

Definition 7 (stratification). *Let* $\mathbf{S} = (\mathbf{Y}, \mathbf{def}, \mathbf{X}, \mathbf{A})$ *be a well-defined schema. A* stratification *of* $\mathbf{S}$ *is a function* $\sigma : \mathbf{Y} \cup \mathbf{X} \to \mathbb{N}$ *such that:*

 - *if there exists a non-empty path from* z *to* z' *in* $D_{\mathbf{S}}$, *then* $\sigma(z) \geq \sigma(z')$;
 - *if there exists in* $D_{\mathbf{S}}$ *a non-empty path from* z *to* z' *that contains an edge corresponding to a negative dependency, then* $\sigma(z) > \sigma(z')$.

It is well known that stratified negation, i.e. no cycles containing negative dependencies, guarantees the existence of a stratification. Note also that w.l.o.g. we can consider that the active range of a stratification is an interval of the form $[1..k]$ for some $k \geq 1$. In order to define $\tau_{\max}$, we introduce a series of typings $\tau_1, \ldots, \tau_k$ such that for every $i \in 1..k$, the typing τ_i contains only shape labels

on stratums j such that $j \leq i$. Each of the intermediate typings is maximal w.r.t. set inclusion, in a sense to be made precise shortly. Then $\tau_{\max} = \tau_k$. We start by introducing some notations. Given a schema $\mathbf{S} = (\mathbf{Y}, \mathbf{def}, \mathbf{X}, \mathbf{A})$ and a stratification σ of $\mathbf{S}$ with active domain $[1..k]$, we denote $\mathbf{Y}_i^\sigma$, resp. $\mathbf{X}_i^\sigma$, resp. $\mathbf{A}_i^\sigma$ the subsets of labels from $\mathbf{Y}$, resp. $\mathbf{X}$, resp. $\mathbf{A}$ whose stratum is at most i. Formally, $\mathbf{Y}_i^\sigma = \{y \in \mathbf{Y} \mid \sigma(y) \leq i\}$, $\mathbf{X}_i^\sigma = \{x \in \mathbf{X} \mid \sigma(x) \leq i\}$ and $\mathbf{A}_i^\sigma = \mathbf{A} \cap \mathbf{X}_i^\sigma$. Additionally, we let $\mathbf{S}_i^\sigma = (\mathbf{Y}_i^\sigma, \mathbf{X}_i^\sigma, \mathbf{def}_i^\sigma, \mathbf{A}_i^\sigma)$ be the schema $\mathbf{S}$ restricted to the labels on stratum at most i. For a typing τ and a set of labels Z we denote by τ_Z the restriction of τ on Z, that is, $\tau_Z = \{(n, z) \in \tau \mid z \in Z\}$.

Definition 8 (maximal typing). *Let* $\mathbf{S} = (\mathbf{Y}, \mathbf{X}, \mathbf{def}, \mathbf{A})$ *be a schema,* $\sigma : \mathbf{Y} \cup \mathbf{X} \to [1..k]$ *be a stratification of* $\mathbf{S}$ *for some* $k \geq 1$, *and* $\mathbf{G}$ *be a graph. For every* $1 \leq i \leq k$, *let* τ_i *be the typing of* $\mathbf{G}$ *by* $\mathbf{S}_i^\sigma$ *defined inductively by:*

- τ_1 *is the union of all correct typings of* $\mathbf{G}$ *by* $\mathbf{S}_1^\sigma$,
- *for every* $1 \leq i < k$, τ_{i+1} *is the union of all correct typings* τ' *which restriction on* $\mathbf{Y}_i^\sigma \cup \mathbf{X}_i^\sigma$ *is* τ_i. *Formally,* $\tau_{i+1} = \bigcup \left\{ \tau' \subseteq \mathcal{C}(\mathbf{G}, \mathbf{S}_{i+1}^\sigma) \mid \tau'_{\mathbf{Y}_i^\sigma \cup \mathbf{X}_i^\sigma} = \tau_i \right\}$.

The typing τ_k *is called the* maximal typing *of* $\mathbf{G}$ *by* $\mathbf{S}$ *with* σ *and is denoted* $\tau_{\max}(\mathbf{S}, \sigma, \mathbf{G})$.

Proposition 1. *For every schema* $\mathbf{S}$, *stratification* σ *of* $\mathbf{S}$, *and graph* $\mathbf{G}$, *the typing* $\tau_{\max}(\mathbf{S}, \sigma, \mathbf{G})$ *is a correct typing.*

The maximal typing can be defined independently on a particular stratification, thanks to the lemma below. Therefore, we denote by $\tau_{\max}(\mathbf{S}, \mathbf{G})$ the unique maximal typing of $\mathbf{G}$ by $\mathbf{S}$.

Lemma 2. *For every schema* $\mathbf{S}$, *every graph* $\mathbf{G}$ *and all two stratifications* σ_1 *and* σ_2 *of* $\mathbf{S}$, *it holds that* $\tau_{\max}(\mathbf{S}, \sigma_1, \mathbf{G}) = \tau_{\max}(\mathbf{S}, \sigma_2, \mathbf{G})$.

5   Validation algorithm

The validation problem for ShEx is, given a graph $\mathbf{G}$, a schema $\mathbf{S}$ and a typing τ of $\mathbf{G}$ by $\mathbf{S}$, determine whether $\tau \subseteq \tau_{\max}(\mathbf{S}, \mathbf{G})$. Intuitively, the latter means that for every $(n, z) \in \tau$, the node n conforms to the shape expression named z. A validation algorithm for ShEx with inheritance can be based on computing the maximal typing, using a standard *refinement* algorithm such as that presented in [4]. It goes as follows. Compute a stratification σ for $\mathbf{S}$, then construct the series of typings τ_i from Def. 8. Each of the τ_i is obtained by a refinement that starts with τ_i^c, a *complete typing for stratum* i, and successively removes from it the pairs that are not conformant. More precisely, $\tau_i^c = \tau_{i-1} \cup \{(n, z) \mid nodes(\mathbf{G}) \times (\mathbf{Y} \cup \mathbf{X}) \mid \sigma(z) = i\}$. Then, if τ is the current version of the progressively refined typing, we repeatedly remove from it pairs $(n, z) \in \tau$ such that $\mathbf{G}, n, \tau \neg \models_{\mathbf{S}} \mathbf{def}(z)$, until the typing becomes correct. This algorithm also works for ShEx with inheritance. As shown in Lemma 1, $\mathbf{G}, n, \tau \models_{\mathbf{S}} \mathbf{def}(z)$ can be effectively computed, so validation is decidable. In [4], the authors also present a recursive

algorithm that computes only a portion of $\tau_{\max}(\mathbf{G}, \mathbf{S})$ that is relevant for the validation task. This algorithm also works for ShEx with inheritance. Because the validation algorithms for ShEx with inheritance are essentially the same as for ShEx 2.1, the effort required to adapt the existing ShEx validators is limited.

The inheritance mechanism does not increase the complexity of validation. As shown in [15], validation of ShEx limited to triple expressions only (i.e. the $M, \tau \models e$ relation) is NP-complete, while the refinement algorithm requires a polynomial number of steps. It is easy to see that computing the truth value of $\mathbf{G}, n, \tau \models_{\mathbf{S}} \mathbf{def}(z)$ is in NP. Therefore

Proposition 2. *Validation for ShEx with inheritance is NP-complete.*

6   Discussion

Comparison with the ShEx specification We use two simplifications. First, in the specification, references and shape expressions are interchangeable, e.g., it is possible to write $@y_1$ and $@y_2$ or $\langle\, p\, @(\text{not } c_1 \text{ or } @x)\,\rangle$. Following [4], we allow references only in triple constraints. This does not modify the expressive power of the language, yet it simplifies its presentation and formalization. Second, in the specification, the sets of labels $\mathbf{Y}$, $\mathbf{X}$ and $\mathbf{A}$ are not explicitly given. The set of extendable labels $\mathbf{X}$ is determined by the use of the extends keyword, and similarly the set of abstract labels $\mathbf{A}$ is determined by the use of abstract.

Inheritance-like features in ShEx 2.1 ShEx 2.1 allows for the expression of inheritance in some cases. If a shape expression needs to be extended with new properties only, then conjunction alone can be used, as in this example:

$$Figure \to \langle\, coord\, @Coord\,\rangle \qquad\qquad Circle \to @Figure \text{ and } \langle\, radius\, @T_{float}\,\rangle$$

A *Circle* is thus a *Figure* that has coordinates and a radius. This works because $\langle\, coord\, @Coord\,\rangle$ and $\langle\, radius\, @T_{float}\,\rangle$ use disjoint sets of properties and do not interact with each other. In a slightly different example, we have products with numeric reviews and want to extend them with text reviews:

$$Product \to \langle\, review\, @T_{int}\, *\,\rangle \qquad MyProduct \to @Product \text{ and } \langle\, review\, @T_{str}\, *\,\rangle$$

MyProduct cannot be satisfied because a value of a *review* property is required to be both integer and string. We actually want to be able to write something like $MyProduct \to @Product; \langle\, review\, @T_{str}\, *\,\rangle$, i.e. use the ';' operator to extend the contents of *Product* with additional properties. This would be possible in ShEx 2.1 if the definition of *Product* is a shape (non-terminal h in Def. 1), which is an important limitation. Furthermore, such a mechanism would not allow for easy restriction of the inherited shape, for instance if *MyProduct* required that numeric reviews have values between 1 and 5. Overall, ShEx 2.1 or mild extensions of it could provide inheritance-like features with the cost of modelling gymnastics.

Design choices We impose a syntactic restriction on extendable shape expressions. They must be conjunctions of a shape and a plain shape expression. That is, we disallow the extension of more general expressions, in particular expressions of the form s_1 or s_2. In what follows, we explain the rationale behind this design choice. On the one hand, the motivating use cases clearly demonstrated the need to extend shape expressions of that form, while the need to extend disjunctions was not supported by the use cases. On the other hand, extending on disjunctions together with multiple inheritance complicates the definition of the satisfaction relation, as illustrated by the following. In the example from Sect. 2, *ColouredCircle* inherits from *Figure* in two different ways. Suppose that the definition was $Figure \rightarrow \langle\, coord\; @CartesianCoord\,\rangle$ or $\langle\, coord\; @PolarCoord\,\rangle$. This raises the question whether we allow a *ColouredCircle* to have both Cartesian and polar coordinates, the one inherited from *ColouredFigure*, the other inherited from *Circle*. Allowing it would be undesirable. Disallowing it would complicate the satisfiability relation as it would require (1) recording which one of the disjuncts (here, Cartesian coordinate and polar coordinate) has been satisfied by each subtype (here, by *ColouredCircle* and by *ColouredFigure*), and (2) checking whether all the subtypes satisfy the same disjunct. We decided to take a conservative approach and limit the shape expressions that can be extended.

Limitations The inheritance mechanism can be unintuitive in cases in which a node did not satisfy a shape expression before it was extended, but does conform after the expression is extended, as for instance node `f1` and shape expression *Circle* from Sect. 2. The language is backward compatible in the positive case, i.e. all nodes that satisfy a shape expression continue to satisfy it if it gets extended. Another possible drawback is the need to know all the descendants of a shape expression in order to validate it. This can limit the modularity of the validation.

Inheritance in SHACL? SHACL is an RDF validation language published as a W3C recommendation [7], and has been formalized in several scientific works [6,11,3], among others. We are not aware of the development of an inheritance mechanism for SHACL. We believe that it would require using conjunction, at least for some use cases. Using the syntax of SHACL from [6], the product example can be captured by:

$$Product \rightarrow\, \geq_0 review.T_{int} \qquad MyProduct \rightarrow\; Product \wedge \geq_0 review.T_{str}$$

Then every *MyProduct* would be a *Product*. The following example cannot be handled in the same way: a *Client* shape expression that has an email needs to be extended to *MyClient* that has an additional email. In ShEx with inheritance it is captured by $Client \rightarrow$ extends $\_\langle\, email\; @T_{str}\,\rangle$, $MyClient \rightarrow$ extends $Client\; \langle\, email\; @T_{str}\,\rangle$. A SHACL *MyClient* defined in a way similar to *MyProduct* above would have only one email.

Related work The semantics of ShEx with inheritance presented here is an extension of that of ShEx 2.1 from [4], e.g. based on the existence of a maximal

typing. Overall, the proof follows the same steps, with additional technicality due to the new satisfiability relation $M, \tau \models s$ on the one hand, and to the fact that checking the satisfiability of a shape with extends requires dereferencing its ancestors (line 18.), on the other hand. The syntax of the language is slightly different from [4]. There, triple constraints use sets of properties instead of a single property, which allows encoding the closed and extra modifiers of a shape within its triple expression. We cannot rely on the same simplification as the closed and extra modifiers of the ancestors are ignored, only the triple expression $ext\text{-}te(x)$ is used for satisfaction (line 12. in Table 2).

The notion of inheritance and its different incarnations has a long tradition in object-oriented programming [16], under various names such as sub-typing, generalization/specialization, etc. Many of these proposals date to the 80s. The Liskov substitution principle was proposed in 1987 as a mechanism for conceptual inheritance [10] and the problem of name-collision in multiple-inheritance systems was already discussed at that time [8]. The notion of inheritance and inclusion polymorphism is a key feature of object-oriented languages [5]. We follow this tradition by allowing a hierarchy of shape expressions which is similar to sub-typing; we avoid the term subtype to prevent confusion with the notion of types identified by `rdf:type` in the RDF context.

Given that ShEx was inspired by XML Schema, the basic functionality of *extends* was also inspired by the extension behaviour in XML Schema [14], but we added multiple-inheritance and removed the requirement of type annotations in data. PG-Schema [2] has been proposed as a schema language for labelled property graphs and includes a notion of inheritance with support for multiple-inheritance. However, it checks that nodes conform exactly with their types, without considering their descendants as in our proposal. Extending ShEx to describe property graphs or RDF-Star has been explored in [9], although that paper does not include the inheritance mechanism presented here.

Conclusions We presented an extension of ShEx with an inheritance mechanism. It is called for by use cases and inspired by inheritance in programming languages. Its semantics is well-founded, and builds upon the semantics of ShEx 2.1, which makes it easy to integrate into existing ShEx validators. The extension has several implementations [1]. The design of the inheritance mechanism required some decisions in order to ensure a good trade-off between keeping its conceptual complexity reasonable, while satisfying the motivating use cases. While our proposal has some limitations, we believe that it would allow for better reusability of ShEx schemas.

Appendix

We view the satisfiability relations from Tables 1, 2 and 3 as boolean functions.

Lemma 1 (proof sketch) The proof goes by induction on the tree of recursive calls resulting from the evaluation of $n, \tau \models \mathbf{def}(z)$, let θ be that tree. In θ, every vertex corresponds to one of the lines 1.–18. of the satisfaction functions in Tables 1, 2 and 3 depending on the function being called in that vertex, and on the syntactic form of the shape or triple expression passed as parameter. Remark that almost all direct recursive calls are made on strict sub-expressions, except for lines 11., 12. and the two recursive calls $M_x, \tau \models \textit{ext-te}(x)$ and $\left(\bigcup_{z \in anc(x)} M_z \right), \tau \models \textit{restr}(x)$ on line 18.. Line 11. directly calls line 17. which in turns makes recursive calls on strict sub-expressions. Line 12. directly calls line 18. that we discuss below. The above two cases from line 18. are those that require attention. We show that in θ, if a vertex is a call $r = n, \tau \models \mathsf{extends}\, X h\mathclose{[}\, \mathsf{and}\, s_1\mathclose{]}$ for line 12., then none of its direct recursive calls mentioned earlier has a descendant equal to r. This is due to the acyclic nature of the extension hierarchy. As a consequence, the tree θ of recursive calls is finite.

Proposition 1 The proof goes by induction on the number of stratums.

Base case There is a unique stratum and the dependency relations $\textit{dep-extra-neg}_{\mathbf{S}}$ and $\textit{dep-shape-neg}_{\mathbf{S}}$ are empty. As a consequence, (*) every sub-expression $\mathsf{not}\, s_1$ that appears in the schema is s.t. s_1 contains no references, and every sub-expression $\mathclose{[}\,\mathsf{closed}\,\mathclose{]}\mathclose{[}\,\mathsf{extra}\, P\mathclose{]}\langle\, e\, \rangle$ is s.t. for every $p\, @u \in tcs(e)$, if u is a reference, then $p \notin P$.

We show that (**) if τ' and τ'' are correct typings, their union $\tau = \tau' \cup \tau''$ is also a correct typing then, because there is a finite number of typings, the union of all correct typings (i.e. $\tau_{\max}$) is also correct. For (**), it is enough to show that if $n, \tau' \models \mathbf{def}(z)^{11}$ evaluates to true, then $n, \tau \models \mathbf{def}(z)$ evaluates to true, for every n and every $z \notin \mathbf{A}$ s.t. either $(n, z) \in \tau$, or there exists $x \in \mathbf{A}$ with $z \in desc(x) \setminus \mathbf{A}$ and $(n, x) \in \tau$. This goes by structural induction on the tree of recursive calls for the evaluation of $n, \tau' \models \mathbf{def}(z)$, let θ be that tree.

Once again, we distinguish the cases by the corresponding line in 1.–18. of the algorithms. Lines 1., 6., 10., 13. are base cases (leaves of θ). Lines 1., 6. and 13. do not rely on the typing, while for line 10., obviously $(n', z') \in \tau' \implies (n', z') \in \tau$ for all n', z'. For the induction case, we consider a vertex $r' = \frac{M}{n}, \tau' \models s$ of θ (where $\frac{M}{n}$ is the node or set of triples parameter of the call), and the following induction hypothesis: (ih) for every r'' strict descendant of r' in θ, if r'' evaluates to true, then r'' with same parameters except for the typing τ' being replaced by τ also evaluates to true. We show that then $\frac{M}{n}, \tau \models s$ evaluates to true.

Lines 2., 11., 12., 3., 7., 8., 14., 15. easily follow from (ih). For lines 4., 5., 18. we can take the same values for the sets M_i, M_x and M and then it again follows from (ih). For lines 9. and 16., the expression of the call is $\mathsf{not}\, s_1$ and then by (*), s_1 does not contain references, so its satisfiability does not depend on the

---

[11] the case for τ'' is symmetric

typing. Therefore, the only challenge is line 17.. So, consider a call $r = M, \tau \models [\,\mathsf{closed}\,][\,\mathsf{extends}\,P\,]\langle\, e\,\rangle$, let $r' = M, \tau' \models [\,\mathsf{closed}\,][\,\mathsf{extends}\,P\,]\langle\, e\,\rangle$, and let:

- $M^e_{\tau'} = \{(n, p, o) \in M \mid \exists p\ @s \in tcs(e).\ o, \tau' \models s\}$, and
- $M^e_{\tau} = \{(n, p, o) \in M \mid \exists p\ @s \in tcs(e).\ o, \tau \models s\}$

be the corresponding sets computed in the calls r' and r, respectively. We show that if r' evaluates to true, then $M^e_{\tau'} = M^e_{\tau}$, which using (ih) directly allows deducing that r evaluates to true.

So, assume that r evaluates to true. Remark first that by (ih) we have $M^e_{\tau'} \subseteq M^e_{\tau}$. Suppose by contradiction that $(n_1, p, n_2) \in M$ and $p\ @s \in tcs(e)$ are s.t. $n_2, \tau \models s$ but $n_2, \tau' \neg \models s$; thus $(n_1, p, n_2) \in M^e_{\tau} \setminus M^e_{\tau'}$. Then by (*), we deduce that s is of the form $@z'$, so $p \notin P$. Then (n_1, p, n_2) belongs to the set $M^{\notin}_{\tau'} = \{(n_1, q, n') \in M \mid q \in props(e)\} \setminus M^e_{\tau'}$, computed during the evaluation of r, so $props(M^{\notin}_{\tau'}) \not\subseteq P$, which contradicts the fact that r evaluates to true.

Induction case Let $i > 1$ be in the range of σ, the induction hypothesis is (IH) for every $j < i$, τ_j is a correct typing. We again show that if τ', τ'' are correct typings for $\mathbf{S}^\sigma_i$, their union $\tau = \tau' \cup \tau''$ is also a correct typing. Thus, we need to show that if $n, \tau' \models \mathbf{def}(z)$ evaluates to true, then $n, \tau \models \mathbf{def}(z)$ evaluates to true, for every n and every $z \notin \mathbf{A}$ s.t. either $(n, z) \in \tau$, or there exists $x \in \mathbf{A}$ with $z \in desc(x) \setminus \mathbf{A}$ and $(n, x) \in \tau$. We assume that $\sigma(z) = i$, otherwise (IH) applies directly. Similarly to the base case above, the proof goes by structural induction on the tree of recursive calls θ for $n, \tau' \models \mathbf{def}(z)$. The challenging cases are lines 9., 16. and 17.. We do not have any more the property (*) from above allowing to easily deal with lines 9. 16.. However, we can show that if θ contains a node with expression $\mathsf{not}\ s_1$, then its evaluation depends only on labels lying on stratums strictly less than i. Moreover, by definition of stratification, all labels that appear in s_1 are necessarily on some stratum that is strictly less than i. Note however that, because of the extension mechanism and line 10., the evaluation of s_1 might depend on labels that are descendants of a label appearing in s_1. Because of line 18., it can also depend on labels that appear in the definitions of ancestors of z. Thanks to the *dep-extends* dependency relation, such labels are also on stratums strictly less than i. We thus show that $\frac{M}{n}, \tau' \models \mathsf{not}\ s_1$ evaluates the same as $\frac{M}{n}, \tau'_{|i-1} \models \mathsf{not}\ s_1$, where $\tau'_{|i-1}$ is the restriction of τ' to labels on stratum at most $i - 1$. Similarly, $\frac{M}{n}, \tau \models \mathsf{not}\ s_1$ evaluates the same as $\frac{M}{n}, \tau_{|i-1} \models \mathsf{not}\ s_1$. By definition we know that $\tau'_{|i-1} = \tau_{|i-1} = \tau_{i-1}$, then $\frac{M}{n}, \tau' \models \mathsf{not}\ s_1$ and $\frac{M}{n}, \tau \models \mathsf{not}\ s_1$ have the same value.

As for line 12., the proof uses the arguments from above regarding negative dependencies and stratification, and the arguments from the same line in the base case.

Lemma 2 (proof sketch) Let σ' be a most refined stratification obtained by taking every strongly connected component of $H_\mathbf{S}$ as a separate stratum. Then for every stratification σ, every stratum of σ is a union of stratums of σ'. Using this property, we can show that for every stratification σ, $\tau_{\max}(\mathbf{S}, \sigma, \mathbf{G}) = \tau_{\max}(\mathbf{S}, \sigma', \mathbf{G})$. It then immediately follows that $\tau_{\max}(\mathbf{S}, \sigma_1, \mathbf{G}) = \tau_{\max}(\mathbf{S}, \sigma_2, \mathbf{G})$.

References

1. ShEx with inheritance: companion webpage. https://github.com/weso/shex_extends_paper_companion
2. Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Green, A., Hidders, J., Li, B., Libkin, L., Marsault, V., Martens, W., Murlak, F., Plantikow, S., Savkovic, O., Schmidt, M., Sequeda, J., Staworko, S., Tomaszuk, D., Voigt, H., Vrgoc, D., Wu, M., Zivkovic, D.: PG-Schema: Schemas for Property Graphs. Proc. ACM Manag. Data **1**(2), 198:1–198:25 (2023). https://doi.org/10.1145/3589778
3. Bogaerts, B., Jakubowski, M., den Bussche, J.V.: SHACL: A Description Logic in Disguise. In: Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13416, pp. 75–88. Springer (2022). https://doi.org/10.1007/978-3-031-15707-3_7
4. Boneva, I., Labra Gayo, J.E., Prud'hommeaux, E.G.: Semantics and Validation of Shapes Schemas for RDF. In: The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10587, pp. 104–120. Springer (2017). https://doi.org/10.1007/978-3-319-68288-4_7
5. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. ACM Comput. Surv. **17**(4), 471–522 (1985). https://doi.org/10.1145/6041.6042
6. Corman, J., Reutter, J.L., Savkovic, O.: Semantics and Validation of Recursive SHACL. In: The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11136, pp. 318–336. Springer (2018). https://doi.org/10.1007/978-3-030-00671-6_19
7. Knublauch, H., Kontokostas, D.: Shapes Constraint Language (SHACL). https://www.w3.org/TR/shacl/ (July 2017), W3C Recommendation
8. Knudsen, J.L.: Name Collision in Multiple Classification Hierarchies. In: ECOOP'88 European Conference on Object-Oriented Programming, Oslo, Norway, August 15-17, 1988, Proceedings. Lecture Notes in Computer Science, vol. 322, pp. 93–109. Springer (1988). https://doi.org/10.1007/3-540-45910-3_6
9. Labra Gayo, J.E.: Extending Shape Expressions for different types of knowledge graphs. In: Joint proceedings of the 3rd International workshop on knowledge graph generation from text (TEXT2KG) and Data Quality meets Machine Learning and Knowledge Graphs, co-located with the Extended Semantic Web Conference. CEUR-WS (2024)
10. Liskov, B.: Keynote address - data abstraction and hierarchy. In: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1987 Addendum, Orlando, Florida, USA, October 4-8, 1987. pp. 17–34. ACM (1987). https://doi.org/10.1145/62138.62141
11. Pareti, P., Konstantinidis, G., Mogavero, F., Norman, T.J.: SHACL Satisfiability and Containment. In: The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference, Athens, Greece, November 2-6, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12506, pp. 474–493. Springer (2020). https://doi.org/10.1007/978-3-030-62419-4_27
12. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.R.: Shape expressions: an RDF validation and transformation language. In: Proceedings of the 10th International Conference on Semantic Systems, SEMANTiCS 2014, Leipzig, Germany, September 4-5, 2014. pp. 32–40. ACM (2014). https://doi.org/10.1145/2660517.2660523

13. Sharma, D.K., Prud'hommeaux, E., Booth, D., Nanjo, C., Jiang, G.: Shape Expressions (ShEx) schemas for the FHIR R5 specification. J. Biomed. Informatics **148**, 104534 (2023). https://doi.org/10.1016/J.JBI.2023.104534
14. Siméon, J., Wadler, P.: The essence of XML. In: Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003. pp. 1–13. ACM (2003). https://doi.org/10.1145/604131.604132
15. Staworko, S., Boneva, I., Labra Gayo, J.E., Hym, S., Prud'hommeaux, E.G., Solbrig, H.R.: Complexity and Expressiveness of ShEx for RDF. In: 18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium. LIPIcs, vol. 31, pp. 195–211. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPICS.ICDT.2015.195
16. Taivalsaari, A.: On the Notion of Inheritance. ACM Comput. Surv. **28**(3), 438–479 (1996). https://doi.org/10.1145/243439.243441
17. Thornton, K., Solbrig, H., Stupp, G.S., Labra Gayo, J.E., Mietchen, D., Prud'hommeaux, E., Waagmeester, A.: Using Shape Expressions (ShEx) to Share RDF Data Models and to Guide Curation with Rigorous Validation. In: The Semantic Web - 16th International Conference, ESWC 2019, Portorož, Slovenia, June 2-6, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11503, pp. 606–620. Springer (2019). https://doi.org/10.1007/978-3-030-21348-0_39