Formalizing the Semantics of ShEx with Lean

Jose Emilio Labra Gayo · Eric Prud'hommeaux

WESO Lab, University of Oviedo, Spain



What is ShEx (Shape Expressions)

ShEx = RDF description and validation language

- Concise and human friendly language, created in 2014
- Goal: Improve the quality of Knowledge Graphs
- Well-founded semantics and support for recursive shapes
- ShEx is ± similar to SHACL

rudof = Rust implementation of ShEx, SHACL, etc.



NOTE

SHACL semantics is informally specified in natural language
Recursion was left as implementation dependent feature

ShEx schema example

ShEx schema

```
prefix : <…>
prefix xsd: <…>

:Attribute {
  :name    xsd:string ;
  :value   xsd:string OR xsd:decimal ;
  :related @:Attribute * ;
}

:Colour @:Attribute AND {
  :name ["colour"]
  :scope xsd:string ;
}

:Radius @:Attribute AND {
  :name ["radius"] ;
  :value xsd:decimal
}
```

RDF data

```
prefix : <http://example.org/>

:c1 :name   "colour" ;
    :scope  "fill"   ;
    :value  "#FF0000";
    :related :r1     .

:r1 :name   "radius" ;
    :value  10.1     .

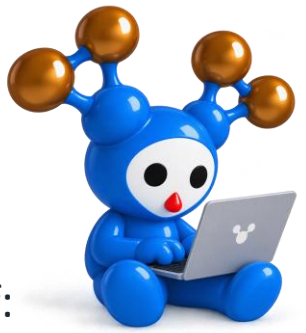
:r2 :name   "radius" ;
    :value  7.2      .

:r3 :name   "radius" ;
    :value  :unknown .

:r4 :name   "radius" ;
    :value  5.0      ;
    :related :r3     .
```

Query map

```
:c1@:Colour,   ✓
:r1@:Radius,   ✓
:r2@:Radius,   ✓
:r3@:Radius,   ✗
:r4@:Radius    ✗
```



Try it in rudof:
https://tinyurl.com/4684vvbw

ShEx specification

Semantics defined in papers:

Semantics and Validation of Shapes Schemas for RDF

I. Boneva, Jose Labra, E. Prud'hommeaux, *ISWC2017*

- Well-founded semantics combines recursion, negation and stratified semantics
- Several complexity results and algorithms
- In the paper, proofs are informally sketched
- Language Spec at https://shex.io/shex-semantics-20170713/
- Contains an extensive test-suite
  - More than one thousand tests

Shape Expressions with Inheritance

I. Boneva, Jose Labra, E. Prud'hommeaux, K. Thornton A. Waagmeester, ESWC 2025

- Updated spec: https://shex.io/shex-semantics/
- Currently, ShEx is under consideration as a IEEE standard



$$\text{se-value-descr1}\ \frac{n \in V}{typing, n \vdash V} \qquad \text{se-value-descr2}\ \frac{_\text{b} \in V \quad n \in \text{Blank}}{typing, n \vdash V}$$

$$\text{se-neig-descr}\ \frac{typing, neigh(n) \models E}{typing, n \vdash E}$$

$$\text{se-shape-and}\ \frac{typing, n \vdash S_1 \quad typing, n \vdash S_2}{typing, n \vdash S_1\ \text{AND}\ S_2} \qquad \text{se-shape-or1}\ \frac{typing, n \vdash S_1}{typing, n \vdash S_1\ \text{OR}\ S_2}$$

$$\text{se-shape-or2}\ \frac{typing, n \vdash S_2}{typing, n \vdash S_1\ \text{OR}\ S_2} \qquad \text{se-shape-not}\ \frac{typing, n \nvdash S}{typing, n \vdash \text{NOT}\ S}$$

(a) Node satisfies a shape expression.

$$\text{te-tpattern}\ \frac{N = \{(subj, pred, obj)\} \quad pred \in P \quad (obj, L) \in typing}{typing, N \models P\ @L} \qquad \text{te-empty}\ \frac{N = \emptyset}{typing, N \models \text{EMPTY}}$$

$$\text{te-some-of1}\ \frac{typing, N \models E_1}{typing, N \models E_1 | E_2} \qquad \text{te-some-of2}\ \frac{typing, N \models E_2}{typing, N \models E_1 | E_2}$$

$$\text{te-each-of}\ \frac{N = N_1 \uplus N_2 \quad typing, N_1 \models E_1 \quad typing, N_2 \models E_2}{typing, N \models E_1; E_2}$$

$$\text{te-repet}\ \frac{N = N_1 \uplus \ldots \uplus N_k \quad min \leq k \leq max \quad typing, N_i \models E \text{ for all } 0 \leq i \leq k}{typing, N \models E[min; max]}$$

(b) Set of triples matches a triple expression.

Fig. 2: Definitions of the $\vdash$ and $\models$ relations.

Motivation

Recently, Lean was applied to prove some mathematical theorems

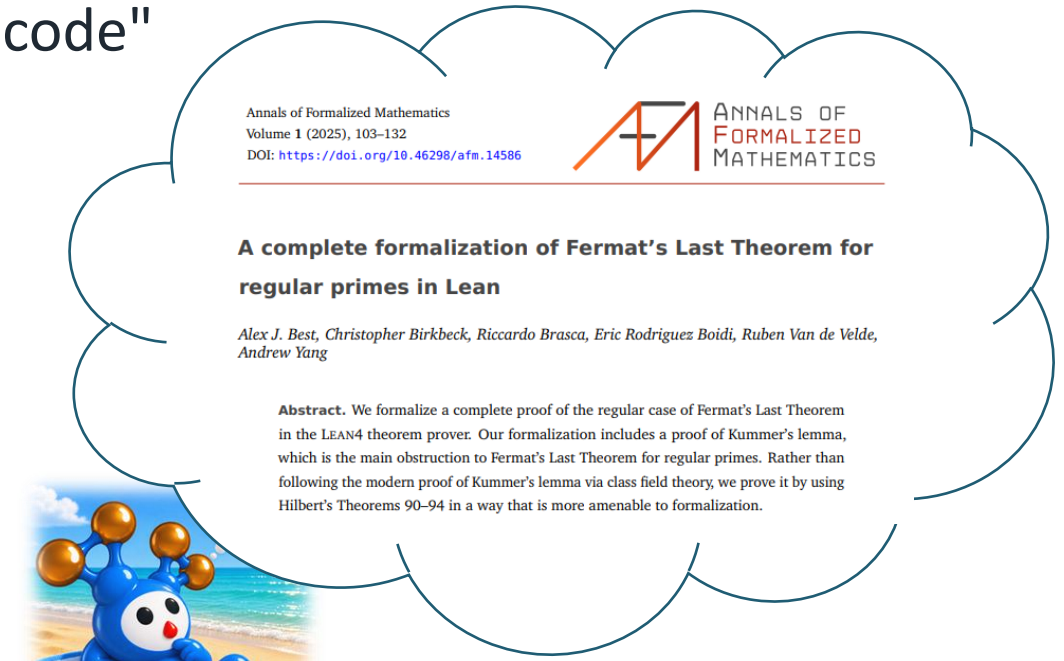
Idea: Can we apply Lean to Semantic Web technologies domain also?

"Summer" exercise:

Apply Lean to verify ShEx semantics: "Proofs in the code"

+

Claude Code to help with encodings and proofs



Contributions



1. Formalization

- A Lean representation of ShEx
- The 2017 semantics: recursion and negation
- The 2025 inheritance semantics: `extends`

2. Machine-checked proofs

- Proofs of the main lemmas and propositions
- Proofs that the validation algorithms calculate the semantics
- A first step to use theorem provers for RDF validation languages

RDF and ShEx in Lean

2017 paper: *Semantics and Validation of Shapes Schemas for RDF*



Lean 4: the basic constructs

- `structure`: a record with named fields
- `inductive`: a type with a full list of constructors
- `abbrev`: a new name for an existing type
- `Finset α` (Mathlib): a finite set of elements of type α

```
structure Triple where
  subj : Subject
  pred : IRI
  obj  : RDFNode
  deriving DecidableEq, Repr

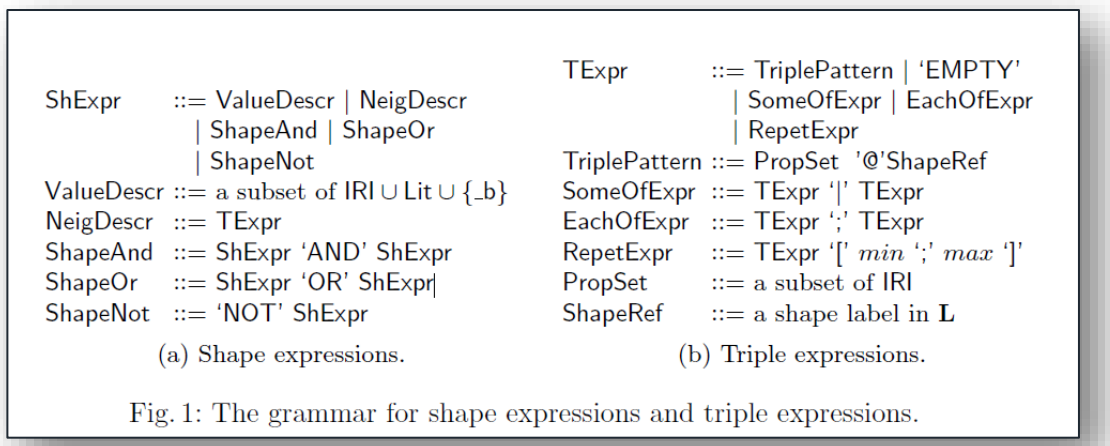
abbrev Graph := Finset Triple

inductive RDFNode where
  | iri   : IRI → RDFNode
  | lit   : Literal → RDFNode
  | bnode : BNodeId → RDFNode
  deriving DecidableEq, Repr

abbrev Typing := Finset Assoc
abbrev Assoc := RDFNode × Label
```

ShEx abstract syntax (2017)

- A schema has a set of labels and a total function `Label → ShExpr`
- Shape and triple expressions are **mutually recursive**
- A shape reference is only a label. The semantics uses a typing to resolve it



ShExpr	::= ValueDescr | NeigDescr | ShapeAnd | ShapeOr | ShapeNot	TExpr	::= TriplePattern | 'EMPTY' | SomeOfExpr | EachOfExpr | RepetExpr
ValueDescr	::= a subset of IRI ∪ Lit ∪ {_b}	TriplePattern	::= PropSet '@'ShapeRef
NeigDescr	::= TExpr	SomeOfExpr	::= TExpr '|' TExpr
ShapeAnd	::= ShExpr 'AND' ShExpr	EachOfExpr	::= TExpr ';' TExpr
ShapeOr	::= ShExpr 'OR' ShExpr	RepetExpr	::= TExpr '[' min ';' max ']'
ShapeNot	::= 'NOT' ShExpr	PropSet	::= a subset of IRI
		ShapeRef	::= a shape label in **L**

(a) Shape expressions. (b) Triple expressions.

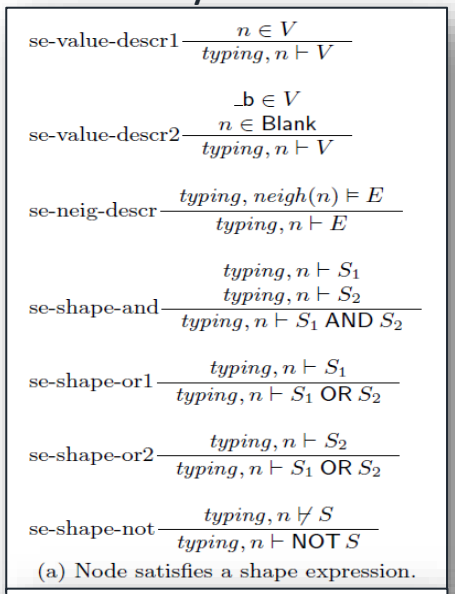
Fig. 1: The grammar for shape expressions and triple expressions.

```
mutual
inductive ShExpr where
  | valueDescr : ValueDescr → ShExpr
  | neigDescr  : TExpr → ShExpr
  | and        : ShExpr → ShExpr → ShExpr
  | or         : ShExpr → ShExpr → ShExpr
  | not        : ShExpr → ShExpr

inductive TExpr where
  | tPattern : PropSet → Label → TExpr
  | empty    : TExpr
  | someOf   : TExpr → TExpr → TExpr
  | eachOf   : TExpr → TExpr → TExpr
  | rep      : TExpr → Nat → Option Nat → TExpr
end
```

Semantics: Satisfies and Matches

Two mutually recursive relations: satisfies (Node - Shape expressions) and matches (neighborhood - triple expressions)



$$\text{se-value-descr1}\ \frac{n \in V}{typing, n \vdash V}$$

$$\text{se-value-descr2}\ \frac{_b \in V \quad n \in \text{Blank}}{typing, n \vdash V}$$

$$\text{se-neig-descr}\ \frac{typing, neigh(n) \models E}{typing, n \vdash E}$$

$$\text{se-shape-and}\ \frac{typing, n \vdash S_1 \quad typing, n \vdash S_2}{typing, n \vdash S_1\ \text{AND}\ S_2}$$

$$\text{se-shape-or1}\ \frac{typing, n \vdash S_1}{typing, n \vdash S_1\ \text{OR}\ S_2}$$

$$\text{se-shape-or2}\ \frac{typing, n \vdash S_2}{typing, n \vdash S_1\ \text{OR}\ S_2}$$

$$\text{se-shape-not}\ \frac{typing, n \nvdash S}{typing, n \vdash \text{NOT}\ S}$$

(a) Node satisfies a shape expression.

$$\text{te-tpattern}\ \frac{N = \{(subj, pred, obj)\} \quad pred \in P \quad (obj, L) \in typing}{typing, N \models P\ @L}$$

$$\text{te-empty}\ \frac{N = \emptyset}{typing, N \models \text{EMPTY}}$$

$$\text{te-some-of1}\ \frac{typing, N \models E_1}{typing, N \models E_1|E_2}$$

$$\text{te-some-of2}\ \frac{typing, N \models E_2}{typing, N \models E_1|E_2}$$

$$\text{te-each-of}\ \frac{N = N_1 \uplus N_2 \quad typing, N_1 \models E_1 \quad typing, N_2 \models E_2}{typing, N \models E_1; E_2}$$

$$\text{te-repet}\ \frac{N = N_1 \uplus \ldots \uplus N_k \quad min \leq k \leq max \quad typing, N_i \models E \text{ for all } 0 \leq i \leq k}{typing, N \models E[min; max]}$$

(b) Set of triples matches a triple expression.

```
def Satisfies (G : Graph) (t : Typing)
              (n : RDFNode) : ShExpr → Prop
  | .valueDescr V => InV n V ∨ BNodeInV n V
  | .neigDescr E => Matches G t (neigh G n) E
  | .and S1 S2 => Satisfies G t n S1 ∧ Satisfies G t n S2
  | .or S1 S2 => Satisfies G t n S1 ∨ Satisfies G t n S2
  | .not S => ¬ Satisfies G t n S
```

```
def Matches (G : Graph) (t : Typing) (N : Finset Triple) : TExpr → Prop
  | .tPattern P L => ∃ subj pred obj,
                     N = {(⟨subj, pred, obj⟩ : Triple)} ∧
                     pred ∈ P ∧
                     t.has obj L
  | .empty => N = ∅
  | .someOf E1 E2 => Matches G t N E1 ∨ Matches G t N E2
  | .eachOf E1 E2 => ∃ N1 N2, DisjointUnion N N1 N2 ∧
                     Matches G t N1 E1 ∧
                     Matches G t N2 E2
  | .rep E min max => ∃ Ns : List (Finset Triple), DisjointUnionList N Ns ∧
      min ≤ Ns.length ∧
      (∀ m', max = some m' → Ns.length ≤ m') ∧
      ∀ Ni ∈ Ns, Matches G t Ni E
```

Stratified negation

With recursion and negation, a schema can have many correct typings.

Solution in ShEx: we don't allow negation in a cycles, i.e. schemas with stratified negation

Definition 3 (schema with stratified negation). *A schema* $\mathbf{S} = (\mathbf{L}, \mathbf{def})$ *is* with stratified negation *if there exists a natural number* k *and a mapping strat from* $\mathbf{L}$ *to the interval* $[1; k]$ *such that for all shape labels* L_1, L_2*:*

- *if* $dep^-(L_1, L_2)$*, then* $strat(L_1) > strat(L_2)$*;*
- *if* $dep^+(L_1, L_2)$*, then* $strat(L_1) \geq strat(L_2)$*.*

The mapping *strat* is called a *stratification* of $\mathbf{S}$. A well known property of stratified negation is that the dependency graph does not have a cycle that goes through a negative dependency edge. This intuitively means that if shape L_1



```
structure Strat (S : Schema) where
  k : ℕ
  str : Label → ℕ
  mem_range : ∀ L ∈ S.labels, 1 ≤ str L ∧ str L ≤ k
  neg_strict : ∀ L1 ∈ S.labels, ∀ L2, DepNeg S L1 L2 → str L2 < str L1
  pos_le : ∀ L1 ∈ S.labels, ∀ L2, DepPos S L1 L2 → str L2 ≤ str L1

/-- `S` is a schema with stratified negation if it has a stratification -/
def Schema.StratNeg (S : Schema) : Prop := Nonempty (Strat S)
```

Lemma 1 and Proposition 1 from paper

Lemma 1: a schema without negation has a maximal correct typing.

Lemma 1. *Let* $\mathbf{S}$ *be a schema that does not use the negation operator* NOT. *Then for all graphs* $\mathbf{G}$, *there exists a correct typing* $typing_g$ *of* $\mathbf{G}$ *by* $\mathbf{S}$ *such that for every* $typing'$, *if* $typing'$ *is a correct typing of* $\mathbf{G}$ *by* $\mathbf{S}$, *then* $typing' \subseteq typing_g$.

```
theorem lemma1 (S : Schema) (hnf : S.NegFree) (G : Graph) :
    ∃ typingG : Typing, typingG ⊆ S.univTyping G ∧ S.CorrTyping G typingG ∧
      ∀ t', t' ⊆ S.univTyping G → S.CorrTyping G t' → t' ⊆ typingG
```

Propposition 1: The stratification-maximal correc typing (Mtyping) is correct

Proposition 1. *For any schema* $\mathbf{S}$, *any stratification* $strat$ *of* $\mathbf{S}$ *and any graph* $\mathbf{G}$, $Typing(\mathbf{G}, \mathbf{S}, strat)$ *is a correct typing of* $\mathbf{G}$ *by* $\mathbf{S}$.

```
theorem proposition1 (S : Schema) (str : Strat S) (hClosed : S.Closed) (G : Graph) :
    S.CorrTyping G (MTyping G S str)
```

- The proof of Proposition 1 uses induction on the strata
- Some implicit declarations were made explicit. For example, declared shapes only reference declared shapes: `hClosed`

Lemma 2: all stratifications give the same typing

Lemma 2. *Let* $\mathbf{S} = (\mathbf{L}, \mathbf{def})$ *be a schema and* $\mathbf{G}$ *be a graph. Let* $strat_1$ *and* $strat_2$ *be two stratifications of* $\mathbf{S}$. *Then* $Typing(\mathbf{G}, \mathbf{S}, strat_1) = Typing(\mathbf{G}, \mathbf{S}, strat_2)$.

- The paper gives only the idea of the proof
- It assumes that a canonical, "most refined" stratification exists

Proof. (Idea) The proof uses a classical technique as e.g. for stratified Datalog. There exists a unique (up to permutation on the numbering of stratums) most refined stratification $strat_{\text{ref}}$ such that for any other stratification $strat'$, each stratum of $strat'$ can be obtained as a union of stratums of $strat_{\text{ref}}$. Then we show that for any stratification $strat'$, $Typing(\mathbf{G}, \mathbf{S}, strat') = Typing(\mathbf{G}, \mathbf{S}, strat_{\text{ref}})$.

```
theorem lemma2 (S : Schema) (hClosed : S.Closed) (str1 str2 : Strat S) (G : Graph) :
    MTyping G S str1 = MTyping G S str2 := by
  rw [← canonicalTyping_eq_MTyping S str1 hClosed G str1,
      ← canonicalTyping_eq_MTyping S str1 hClosed G str2]
```

Lean has the full proof (Lemma2.lean)

- It follows the spirit of the paper: compare everything to one reference, and use the fact that merging strata is harmless, but…
  - Replaces "*most refined stratification*" with a confluence argument (every stratification gives the same MTyping)
  - And it proves the reordering step that the paper leaves out.

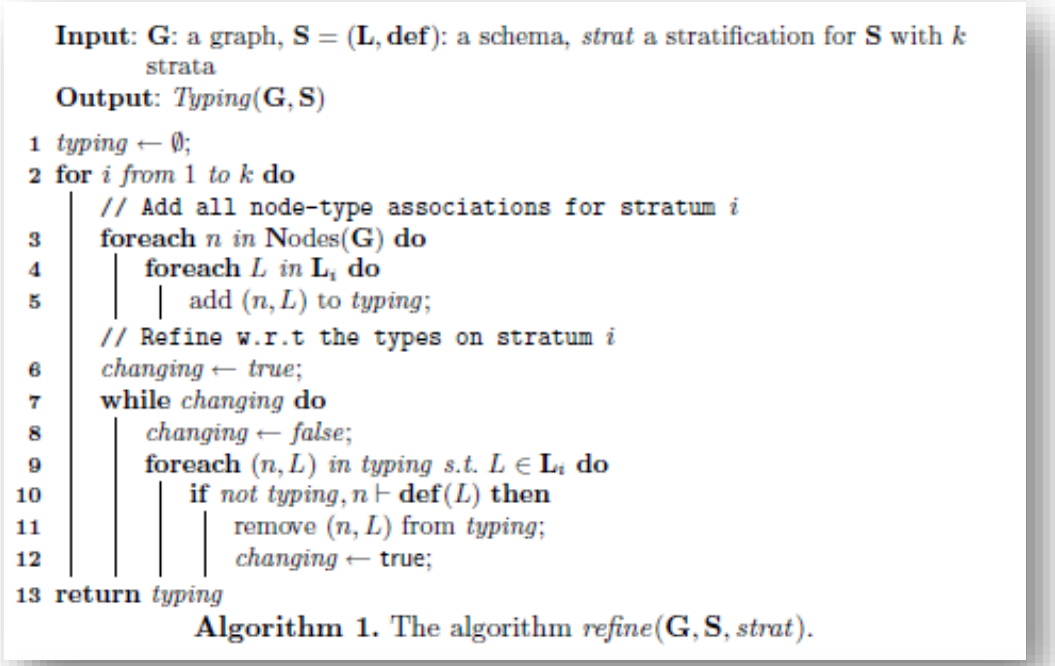
Validation algorithms are correct

Refine calculates the typing, one stratum at a time.

Prove gives the result for one pair (n, L)

```
theorem algorithm1_correct (G : Graph) (S : Schema)
  (str : Strat S) (hClosed : S.Closed) :
 Algorithm1 G S str = MTyping G S str := ...

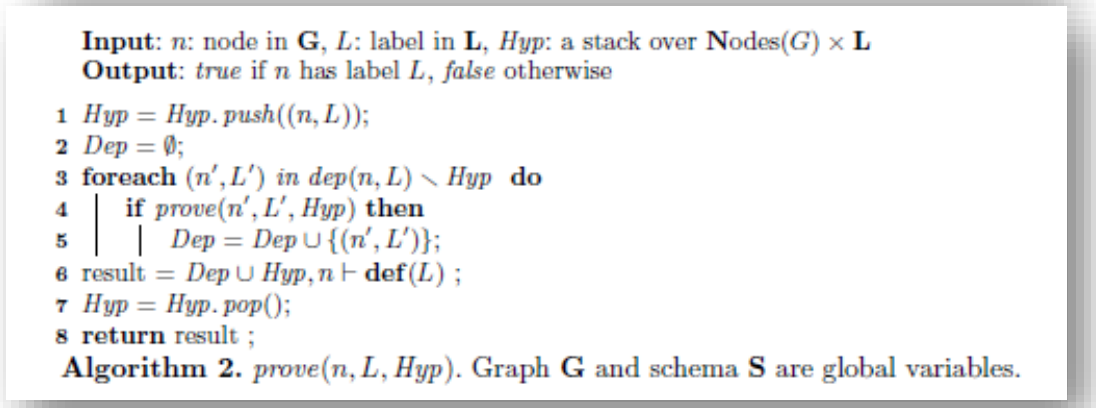
theorem prove_correct (S : Schema)
  (str : Strat S) (hClosed : S.Closed) (G : Graph)
  (n : RDFNode) (L : Label) (hLlabels : L ∈ S.labels) :
    prove S G n L = true ↔ (n, L) ∈ MTyping G S str
```



Input: $\mathbf{G}$: a graph, $\mathbf{S} = (\mathbf{L}, \mathbf{def})$: a schema, *strat* a stratification for $\mathbf{S}$ with k strata
Output: $Typing(\mathbf{G}, \mathbf{S})$

```
1  typing ← ∅;
2  for i from 1 to k do
       // Add all node-type associations for stratum i
3      foreach n in Nodes(G) do
4          foreach L in L_i do
5              add (n, L) to typing;
       // Refine w.r.t the types on stratum i
6      changing ← true;
7      while changing do
8          changing ← false;
9          foreach (n, L) in typing s.t. L ∈ L_i do
10             if not typing, n ⊢ def(L) then
11                 remove (n, L) from typing;
12                 changing ← true;
13 return typing
```

Algorithm 1. The algorithm *refine*$(\mathbf{G}, \mathbf{S}, strat)$.



Input: n: node in $\mathbf{G}$, L: label in $\mathbf{L}$, Hyp: a stack over $\mathbf{Nodes}(G) \times \mathbf{L}$
Output: *true* if n has label L, *false* otherwise

```
1  Hyp = Hyp.push((n, L));
2  Dep = ∅;
3  foreach (n', L') in dep(n, L) \ Hyp do
4      if prove(n', L', Hyp) then
5          Dep = Dep ∪ {(n', L')};
6  result = Dep ∪ Hyp, n ⊢ def(L) ;
7  Hyp = Hyp.pop();
8  return result ;
```

Algorithm 2. *prove*(n, L, Hyp). Graph $\mathbf{G}$ and schema $\mathbf{S}$ are global variables.

- `prove` accepts a pair that is already on the call stack (coinductive style, greatest fixpoint semantics in each stratum)
- We use a stronger, neighbour-local version of `Satisfies` and `Matches`

Adding inheritance

2025 paper: "*Shape expressions with inheritance*"

Reuse Shapes by extending base shapes

Several features like:

- Multiple inheritance
- Abstract shapes
- . . .



Example with inheritance

ShEx schema

```
abstract :Figure {
  :attr  @:PosX ;
  :attr  @:PosY
}
:Circle extends @:Figure {
  :attr @:Radius
}
:Coloured extends @:Figure {
  :attr @:Colour
}
:ColouredCircle extends @:Circle
                extends @:Coloured { }

:PosX {
 :name  [ "posX" ]  ;
 :value xsd:decimal
}
:PosY {
 :name  [ "posY" ]  ;
 :value xsd:decimal
}
```

RDF data

```
. . . # previous data

:x1 :name  "posX"    ;
    :value 2.0       .

:y1 :name  "posY"    ;
    :value 1.5       .

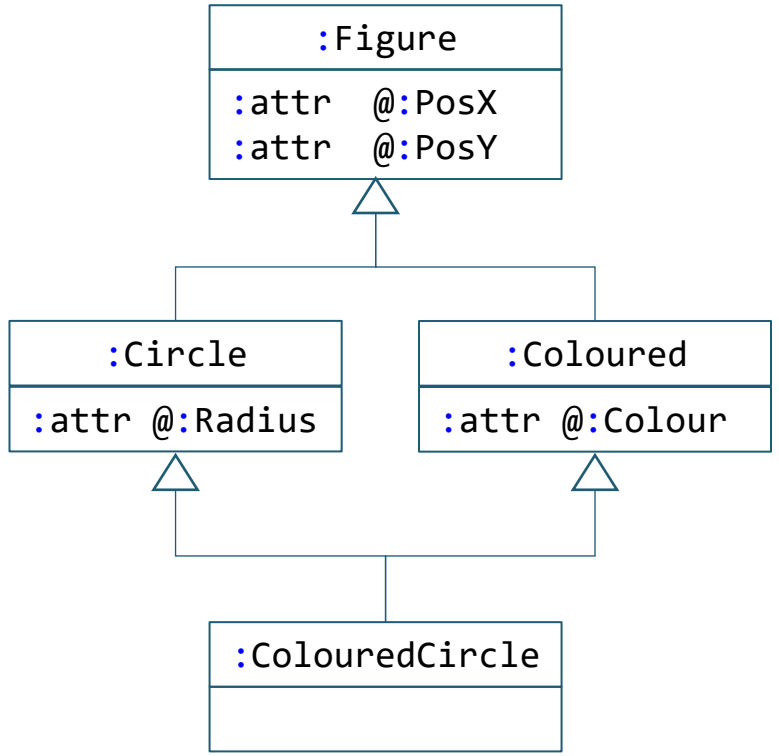
:x2 :name  "posX"    ;
    :value 1.0       .

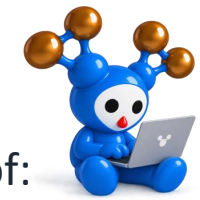
:y2 :name  "posY"    ;
    :value 2.0       .

:fig1 :attr  :r1, :c1, :x1, :y1 .
:fig2 :attr  :r2, :x2, :y2 .
```

Query map

```
:fig1@:ColouredCircle,
:fig2@:Circle
```





Try it in rudof:
https://tinyurl.com/mpv22yt7

Abstract syntax of ShEx with inheritance

$u ::= h \mid t \mid c \mid u \text{ or } u \mid u \text{ and } u \mid \text{not } u$. *(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 IRI ∪ Blank ∪ Literal* . *(node constraint)*

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

```
inductive ShapeExpr where
  | ofShape (h : Shape) : ShapeExpr
  | extends (xs : Finset Label) (h : Shape) : ShapeExpr
  | constraint (c : NodeConstraint) : ShapeExpr
  | or (s1 s2 : ShapeExpr) : ShapeExpr
  | and (s1 s2 : ShapeExpr) : ShapeExpr
  | not (s : ShapeExpr) : ShapeExpr
```

```
structure Shape where
  closed : Bool
  extra  : Option (Finset IRI)
  triples : TripleExpr
```

```
inductive TripleExpr where
  | empty : TripleExpr
  | tc (p : IRI) (z : Label) : TripleExpr
  | choice (e1 e2 : TripleExpr) : TripleExpr
  | seq (e1 e2 : TripleExpr) : TripleExpr
  | star (e : TripleExpr) : TripleExpr
```

- A shape can extend labels with `extends`
- The `closed` and `extra` modifiers are part of the syntax

Semantics of ShEx with inheritance

`satisfiesE` = satisfaction relation on triple expressions

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*	$\exists k \in 0..\lvert M\rvert.\ \exists M_1, \ldots, M_k.$ $M = M_1 \uplus \cdots \uplus M_k \quad \wedge \quad \forall i \in 1..k.\ M_i, \tau \models e_1$

```
def SatisfiesE (τ : Typing) : Finset Triple → TripleExpr → Prop
  | M, .empty => M = ∅
  | M, .tc p z => ∃ n o, M = {(⟨n, p, o⟩ : Triple)} ∧ τ.has o z
  | M, .choice e1 e2 => SatisfiesE τ M e1 ∨
                        SatisfiesE τ M e2
  | M, .seq e1 e2 => ∃ M1 M2, DisjointUnion M M1 M2 ∧
                     SatisfiesE τ M1 e1 ∧
                     SatisfiesE τ M2 e2
  | M, .star e => ∃ Ns : List (Finset Triple),
                  DisjointUnionList M Ns ∧
                  ∀ Ni ∈ Ns, SatisfiesE τ Ni e
```

Semantics of ShEx with inheritance

`satisfiesAt` = satisfaction relation on nodes and shape expressions

The `satM` parameter is included to keep Table 2 independent

Open Recursion Pattern (taking the recursive call as an input)

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.	[closed] [extra P] $\langle e \rangle$	$neigh_{\mathbf{G}}(n), \tau \models$ [closed] [extra P] $\langle e \rangle$
12.	extends X h	$neigh_{\mathbf{G}}(n), \tau \models$ extends X h



```
def SatisfiesAt (G : Graph)
                (satM : Finset Triple → ShapeExpr → Prop)
                (n : RDFNode) : ShapeExpr → Prop
  | .constraint c => c n
  | .or s1 s2 => SatisfiesAt G satM n s1 ∨
                 SatisfiesAt G satM n s2
  | .and s1 s2 => SatisfiesAt G satM n s1 ∧
                  SatisfiesAt G satM n s2
  | .not s => ¬ SatisfiesAt G satM n s
  | .ofShape h => satM (neigh G n) (.ofShape h)
  | .extends xs h => satM (neigh G n) (.extends xs h)
```

Semantics of ShEx with inheritance

`satisfiesMAt` = satisfaction relation on set of triples and shape expressions

The `restr` follows also the open recursion pattern

`satisfiesMFuel` was added to guarantee termination because rule 18 was not structural

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

```
def SatisfiesMAt (S : Schema) (τ : Typing)
                 (evalRestr : Finset Triple → ShapeExpr → Prop) :
    Finset Triple → ShapeExpr → Prop
  | M, .constraint c => c (subjectOf M)
  | M, .or s1 s2 => SatisfiesMAt S τ evalRestr M s1 ∨
                    SatisfiesMAt S τ evalRestr M s2
  | M, .and s1 s2 => SatisfiesMAt S τ evalRestr M s1 ∧
                     SatisfiesMAt S τ evalRestr M s2
  | M, .not s => ¬ SatisfiesMAt S τ evalRestr M s
  | M, .ofShape h => SatisfiesShape τ M h
  | M, .extends xs h =>
      ∃ M' Mx, DisjointUnionOn M M' (S.ancSet xs) Mx ∧
        SatisfiesShape τ M' h ∧
        (∀ x ∈ S.ancSet xs, SatisfiesE τ (Mx x) (S.shapeDef x).extTe) ∧
        (∀ x ∈ S.ancSet xs, ∀ u, (S.shapeDef x).restrOf = some u →
          evalRestr ((S.anc x).biUnion Mx) u)

def SatisfiesShape (τ : Typing) (M : Finset Triple) (h : Shape) : Prop :=
  SatisfiesE τ (Me τ M h.triples) h.triples ∧
  props (Mnotin τ M h.triples) ⊆ h.extra.getD ∅ ∧
  (h.closed → props M ⊆ h.triples.props ∪ h.extra.getD ∅)

def SatisfiesMFuel : (fuel : ℕ) → Schema → Typing → Finset Triple → ShapeExpr → Prop
  | 0, S, τ => SatisfiesMAt S τ (fun _ _ => False)
  | fuel + 1, S, τ => SatisfiesMAt S τ (SatisfiesMFuel fuel S τ)
```

Formalized results and gaps

Result	2017	Inheritance	Gap
Lemma 1 (maximal typing, no negation)	✓	–	–
Proposition 1	✓	✓	–
Lemma 2	✓	✓	**Paper: only a sketch;** **Lean: full proof**
Algorithm 1 (refine)	✓	✓	–
Algorithm 2 (prove)	✓	✓ NEW	**Paper: sketch covers only termination** **Lean: proves correctness**
Proposition 2 (complexity)	Paper: no claim; Lean: NP-hard and coNP-hard NEW	Paper: claimed NP-complete; Lean: coNP-hard with `not` NEW	**false with negation (unless NP = coNP);** **NP-complete without not and restrictions**

NEW = work done after the paper was submitted

Conclusions

- Lean can check the semantics of ShEx and its proofs
- Formalization helped identify implicit hypotheses, incomplete proofs and some gaps
- Generative AI (Claude Code) helps writing the proofs and encodings

Future work

- Add Algorithm 2 to the inheritance formalism (done ✓)
- Explore the design space
  - Inheritance of disjunctive shapes (done ✓)
  - Improve performance (parallel validation, subsets)
  - ShEx for RDF 1.2
- Similar experiment for SHACL
  - See recent paper: *Common foundations of recursive shapes*
- Make this kind of work more reusable
  - A library of Lean tactics for Semantic Web languages?



Open repo: https://github.com/weso/shex_lean

Thank you

Questions?

ShEx in Lean: https://github.com/weso/shex_lean

Rudof:

