Formalizing the Semantics of ShEx with Lean

Jose Emilio Labra Gayo[1][0000−0001−8907−5348] and Eric Prud'hommeaux[0000−0003−1775−9921]

[1] WESO Lab, University of Oviedo, Spain, labra@uniovi.es
[2] eric@uu3.org

Abstract. Validation of RDF data is a key task for the Semantic Web and Knowledge Graphs. Shape Expressions (ShEx) is a popular language for RDF validation that has been presented with a formal semantics in several academic papers. We present a Lean 4 representation of ShEx based on two papers with machine-checked proofs of the main semantic results and of the validation algorithms presented in those papers, within a simplified model of RDF. The development of this encoding can help to extend the language with new features.

Keywords: RDF · ShEx · Shape Expressions

1 Introduction

RDF is the data model underlying the Semantic Web and Knowledge Graphs. Describing and validating RDF data has received growing attention over the last decade, with two main technologies proposed: SHACL [9] and ShEx [17]. SHACL (Shapes Constraint Language) is a W3C recommendation whose definition was based on an informal semantics in natural language that left several features like recursion undefined. On the other hand, ShEx followed a community-driven approach [16] whose semantic specification was based on the formal semantics presented in several papers: [18] introduced the complexity of the basic validation algorithm, [4] presented the semantics of the ShEx schema language with a formal semantics that supported recursion and negation based on stratified semantics, and [5] extended the semantics to support inheritance in ShEx schemas.

Lean 4 [14] has been introduced as an interactive theorem prover based on dependent type theory that can be used to verify the correctness of mathematical proofs. In this paper we present a Lean 4 representation of the ShEx language based on the formal semantics presented in [4] and [5]. Our goal is to verify the correctness of the lemmas, propositions and algorithms presented in those papers. This exercise is a first step toward evaluating the suitability of interactive theorem provers for verifying the semantics of RDF validation languages. The proofs and definitions in Lean follow those given in the original papers.

This paper has two main contributions. The first is a Lean 4 formalization of the ShEx language. The second is a set of machine-checked proofs that the validation algorithms of [4] and [5] compute this semantics.

2 Related Work

Interactive theorem provers such as Lean 4 [14], Rocq (formerly Coq) [20] and Isabelle/HOL [15] allow definitions and proofs to be stated in a formal language and machine-checked, rather than only reviewed by hand. They have been used to formalize substantial bodies of mathematics and, increasingly, the semantics of programming languages and logics used in practice [7], as illustrated by verified systems such as the CompCert compiler [12,13], the seL4 microkernel [8], and by more recent industrial applications such as AWS's Cedar authorization language, formalized in Lean [6].

The application of these techniques to Knowledge Graphs and the Semantic Web is still less mature than in other branches of mathematics. Some earlier work used Isabelle/HOL to formalize OWL semantics in [21], and [19] verifies Datalog using Lean.

ShEx was initially introduced in [17] in an informal way based on an implementation. Later, [18] established the complexity of Shape Expressions without negation and presented a formal semantics for them based on regular bag expressions. An implementation based on regular expression derivatives was proposed in [11]. The formal semantics of the ShEx language including negation and recursion was presented in [4] and published as a W3C Community Group Report [16]. More recently, [5] presented an extension of ShEx with inheritance, which is currently developed by the IEEE P3330 Shape Expression Schemas Working Group. Subsequent work has compared ShEx with other validation approaches, such as SHACL [9] and PGSchema [3], and formalized the common foundations of the three languages: [1] compares ShEx, SHACL and PGSchema without considering recursion, and [2] compares ShEx and SHACL with recursion. None of these works employed an interactive theorem prover to verify the correctness of their semantics and algorithms.

3 Representing RDF data in Lean

This section introduces the Lean declarations used throughout the paper. Lean 4 provides three constructs for defining types, all used below: `structure` defines a record type, a fixed set of named fields grouped into one value (analogous to a class with only data members); `inductive` defines a type by an exhaustive list of constructors, each optionally carrying arguments, so a value of the type is built from exactly one of the listed cases (this is how ShEx's abstract syntax and semantic rules are represented as data, in Sections 4 and 5); and `abbrev` introduces a transparent name for an existing type, used purely for readability. `Finset` α, from Lean's Mathlib library, is the type of finite sets of elements of type α.

We define an RDF graph as a finite set of RDF triples:

```
1 abbrev Graph := Finset Triple
```

where a `Triple` is a structure:

```
structure Triple where
  subj : Subject
  pred : IRI
  obj  : RDFNode
```

such that an `RDFNode` is either an IRI, a literal or a blank node:

```
inductive RDFNode where
  | iri   : IRI → RDFNode
  | lit   : Literal → RDFNode
  | bnode : BNodeId → RDFNode

inductive Subject where
  | iri   : IRI → Subject
  | bnode : BNodeId → Subject
```

where IRI, Literal and BNodeId have a simplified definition as:

```
abbrev IRI := String
abbrev BNodeId := String
inductive Literal where
  | str : String → Literal
  | int : Int → Literal
```

A typing is a finite set of pairs of RDF nodes and shape labels:

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

4 Representing ShEx abstract syntax and typings in Lean

The ShEx abstract syntax from [4] is represented in Lean as follows. A ShEx schema is a finite set of shape labels and a function that maps each shape label to a shape expression:

```
structure Schema where
  labels   : Finset Label
  shapeDef : Label → ShExpr
```

A shape expression `ShExpr` is either a value description, a triple expression, or a boolean combination of shape expressions, and a triple expression `TExpr` is either a triple pattern, the empty triple expression, a choice or conjunction of triple expressions, or a repetition of a triple expression. Since a shape expression's neighbourhood description embeds a triple expression, and a triple pattern refers back to a shape label, the two are declared as a single mutually recursive block:

```
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
```

A value description `ValueDescr` is a finite set of `ValueItems` (IRIs, literals or any blank node), and a `PropSet` is a subset of properties (IRIs).

The encoding simplifies [4,5] in a few points that do not affect the proofs: IRIs, blank nodes and labels are strings, and literals are restricted to strings and integers; `shapeDef` is total, so closure of the schema (`Closed`) is an explicit hypothesis; in the inheritance formalism, acyclicity of **`extends`** (`ExtAcyclic`), invariant (*) (`WellFormedExtra`) and the absence of **`extends`** under `not` (`NotIsExtendsFree`) are also explicit hypotheses.

5 Representing semantic rules

The basic semantics of ShEx formalizes the `Satisfies` (⊢) and `Matches` (⊨) relations of Fig. 2 in [4], both parameterized by a typing `t`. The two relations are mutually recursive: a shape expression's neighbourhood description invokes `Matches` on the node's outgoing triples, and a triple expression's shape reference invokes `Satisfies` indirectly, through membership in `t`. Because each relation calls the other, they are declared together as a single **`mutual`** block, mirroring the paper's joint Fig. 2a/2b presentation.

Although `Satisfies` and `Matches` look like a classic mutually inductive derivability relation, they are encoded as ordinary **`Prop`**-valued recursive functions rather than a mutual inductive family (an inductive family would declare each rule as a constructor of the relation, rather than as a case of a function returning that relation). This is necessary because the premise in the rule for negation uses `Satisfies` negatively, i.e. under a ¬: Lean's positivity checker requires every recursive occurrence in an inductive definition to appear only positively (never to the left of an implication or under a ¬), since otherwise the definition could be used to derive a contradiction. Ordinary recursive functions are not subject to this restriction, so defining `Satisfies` as a function sidesteps it.

`Satisfies` and `Matches` are defined case by case, using Lean's equational function syntax: each `| .constructor args ⇒ body` line supplies the definition for one constructor of the input type. Each such equation below corresponds to one (or a disjunction of two) rules from the paper.

```
mutual
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
end
```

`InV` is a proposition that passes if `n` is an RDF Node in `V`, `BNodeInV` passes if `n` is a blank node in `V`, and `neigh` is a function that returns the set of triples in `G` that have `n` as subject. `DisjointUnion` and `DisjointUnionList` are defined as follows:

```
def DisjointUnion (N N1 N2 : Finset Triple) : Prop :=
  Disjoint N1 N2 ∧ N = N1 ∪ N2

def DisjointUnionList (N : Finset Triple)
                      (Ns : List (Finset Triple)) : Prop :=
  Ns.Pairwise Disjoint ∧ N = Ns.foldr (· ∪ ·) ∅
```

6 Proving lemmas and propositions

6.1 Stratified schemas

Mixing unrestricted recursion with negation can make *correct typing* fail to find a unique semantics: the same schema can admit several incompatible correct typings at once. A standard fix, in the spirit of stratified negation in logic programming, is to forbid negation from ever occurring on a recursive cycle between shape labels. This is captured by a dependency graph on shape labels with two kinds of edges: `DepPos S L1 L2` holds when the shape label L2 occurs as a shape reference in L1's definition without being under a NOT (a positive dependency), and `DepNeg S L1 L2` holds when L2 occurs under some NOT (a negative dependency).

```
abbrev DepPos (S : Schema) (L1 L2 : Label) : Prop := L2 ∈ (S.shapeDef L1).posDeps

abbrev DepNeg (S : Schema) (L1 L2 : Label) : Prop := L2 ∈ (S.shapeDef L1).negDeps
```

A schema with stratified negation admits a stratification `Strat`, represented as a structure that packages the number of strata `k`, the stratum `str` of each label, and proofs that this assignment respects the schema's dependency graph: negative dependencies must strictly decrease the stratum (`neg_strict`), while positive dependencies may never increase it (`pos_le`).

```
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 admits a stratification, i.e. if some term of type `Strat S` exists; Lean's `Nonempty T` expresses exactly this "a witness of type `T` exists" without naming one:

```
def Schema.StratNeg (S : Schema) : Prop := Nonempty (Strat S)
```

6.2 Maximal correct typing

Lemma 1 covers the base case where negation is absent: for a negation-free schema `S` (`Schema.NegFree`: every label's definition uses no NOT) and any graph `G`, there is a correct typing that contains every other correct typing of `G` by `S`. The reason this case is easy is monotonicity: without negation, `Satisfies` only grows with the typing, a node satisfying a shape expression under some typing keeps satisfying it under any larger one, so simply taking the union of all correct typings (`Schema.maximalTyping`) is itself correct, and contains every other one by construction.

```
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
```

It is possible to construct a stratified-maximal correct typing `MTyping` one stratum at a time.

```
def MTyping (G : Graph) (S : Schema) (str : Strat S) : Typing :=
  typ S str.str G str.k

def typ (S : Schema) (str : Label → ℕ) (G : Graph) : ℕ → Typing
  | 0 ⇒ ∅
  | i + 1 ⇒ maxAgree (S.stratRestrict str (i + 1)) G
                     (S.stratum str i) (typ S str G i)
```

where `maxAgree S' G Lo fixed` is the union of all correct typings of `G` by `S'` that agree with a given typing fixed on the labels in `Lo` — the per-stratum construction that `typ`'s inductive step performs, generalizing Lemma 1's maximal correct typing (the special case `Lo = ∅`, no agreement constraint).

For any schema `S`, any stratification `str` of `S` and any graph `G`, the `MTyping G S str` is a correct typing of `G` by `S`.

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

The Lean proof strengthens the statement before inducting: rather than reasoning about MTyping directly, it shows by induction on the stratum index i that `typ S str G i` satisfies two invariants at once — it stays within S_i's universe of typings, and it is already a correct typing of S_i. The base case ($i = 0$) is vacuous, since the initial stratum L_0 is empty. The inductive step reduces to a single correctness fact about `maxAgree` (the union-of-typings construction underlying `typ`), applied with the previous stratum as the "agreement set" — well-formedness of the schema together with the stratification's `neg_strict` condition is exactly what guarantees that S_{i+1}'s negative dependencies stay confined there. Proposition 1 is then just this invariant read off at the last stratum, $i = k$.

`hClosed` is a well-formedness hypothesis with no counterpart in the paper's own statement of Proposition 1: it requires every shape label referenced inside `S`'s definitions to itself be declared. In the paper this holds automatically, since schemas are only ever defined on their own label set; in Lean, `shapeDef` must be a total function, so this closure condition has to be assumed explicitly.

Lemma 2 states that for any two stratifications `str1`, `str2` of a schema `S` the maximal correct typing does not depend on which stratification was chosen:

```
theorem lemma2 (S : Schema) (hClosed : S.Closed)
               (str1 str2 : Strat S)
               (G : Graph) : MTyping G S str1 = MTyping G S str2
```

Here the paper and the Lean development diverge more than for Proposition 1: the paper's own proof is stated only as an idea, asserting without further justification that a canonical *most refined* stratification exists which every other stratification agrees with. The Lean proof has to supply this argument in full. It introduces a canonical process that starts from the empty typing and repeatedly reveals the least-stratum batch of labels still remaining; an *Independence* argument then shows, by strong induction on the number of labels left, that the process's final result does not depend on the order in which batches are revealed — the technical heart being a *Swap* lemma letting two mutually-independent batches be revealed in either order. Running this process along the checkpoints of an arbitrary stratification shows it agrees with `MTyping` for that stratification; applying this to `str1` and `str2` separately, both equal the same canonical value, giving Lemma 2.

7 Validation algorithms

The previous definitions characterize the correct semantics of a schema declaratively as the union of correct typings that a stratification happens to pick out, but say nothing about how to compute it. The paper gives two validation algorithms and states their correctness with respect to that declarative semantics: Algorithm 1 (*refine*), which computes the whole typing `MTyping G S str` at once, and Algorithm 2 (*prove*), which decides membership of a single node-label pair without ever constructing the whole typing. Both algorithms, and both correctness statements, have been formalized and machine-checked in Lean (as mathematical functions; executable checking of the examples uses the separate decision procedures in `Decidable.lean`).

7.1 Algorithm 1: refine

Refine computes one stratum at a time: for stratum i, it starts from the largest plausible typing (everything already fixed for stratum $i-1$, plus every possible guess for the new labels of stratum i), then repeatedly strips out any node-type association that currently fails `Satisfies`, until nothing more can be removed — a greatest-fixed-point computation, one per stratum, chained together.

```
def Algorithm1 (G : Graph) (S : Schema) (str : Strat S) : Typing :=
  refineAlg S str.str G str.k

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

The Lean proof mirrors this stratum-by-stratum structure directly. A single refinement pass (`refineStep`) is iterated to a fixed point (`refine1`), and a core lemma (`refine1_eq_maxAgree`) shows that this iterative fixed point computes exactly the same typing as `maxAgree`, the union of correct typings construction that `typ`'s own inductive step uses. Chaining this equivalence across all k strata by induction gives `algorithm1_correct`: two different-looking constructions, iterative refinement versus a declarative union, turn out to compute the same object at every stratum.

7.2 Algorithm 2: prove

Prove decides whether (n, L) belongs to the typing without ever constructing it, by recursively visiting only the portion of the dependency graph reachable from (n, L), tracking a stack of node-label pairs currently being resolved (to detect and trust positive cycles) and a set `Dep` of pairs it has confirmed along the way.

```
def prove (S : Schema) (G : Graph) (n : RDFNode) (L : Label) : Bool :=
  proveAux S G (G.nodes ×ˢ S.labels) n L ∅

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
```

This proof is structurally the more delicate of the two. `proveAux` is a *coinductive-style* algorithm — when it re-encounters a pair already on the call stack, it simply trusts it, which is exactly what makes positive recursive cycles terminate — so relating it to `MTyping`'s greatest-fixed-point characterization needs a genuine witness typing, not just an unfolding argument. The proof first strengthens the semantic relations `Satisfies`/`Matches` to a *neighbor-local* form, showing that whether (n, L) holds depends on the ambient typing only through its restriction to (n, L)'s own dependency set — the fact that lets `prove` get away with tracking `Dep`/`Hyp` instead of a whole typing. Soundness and completeness are then proved together, by strong induction on the stratum, as a single strengthened statement (`MainProp`) relating `proveAux`'s call-stack bookkeeping to membership in `typ` at that stratum:

```
def MainProp (S : Schema) (str : Strat S) (G : Graph) (U : Finset Assoc)
    (i : ℕ) : Prop :=
  ∀ Hyp : Finset Assoc,
    (∀ p ∈ Hyp, p.2 ∈ S.stratum str.str i → p ∈ typ S str.str G i) →
    ∀ n L, str.str L = i → L ∈ S.labels →
      (proveAux S G U n L Hyp = true ↔ (n, L) ∈ typ S str.str G i)
```

The full argument runs several hundred lines in `Algorithm2.lean`; we refer to it, and to §4.2 of the paper for the algorithm's original presentation, rather than reproduce it here.

8 Adding inheritance

The latest version of ShEx added the notion of shape *inheritance* and was presented in [5]: a shape can extends one or more other (abstract) labels, and valida-

tion against an abstract label succeeds if the node matches *some* concrete descendant of it. Rather than retrofit this onto the 2017 formalization, we ported it as a genuine second formalism, `ShexLean.Inheritance`, with its own `ShapeExpr`/`Schema`, reusing some common parts like (`Graph`, `Typing` and `Label`). The syntax itself grows in two ways beyond what 2017 needed: node constraints become an arbitrary boolean predicate rather than a finite `ValueDescr`, and `closed`/`extra` shape modifiers and `extends` become first-class:

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

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
```

A `Shape`'s `triples` field is itself a triple expression, `TripleExpr`, the inheritance formalism's counterpart to `TExpr` from Section 4: the empty expression, a single triple constraint (property `p`, target label `z`), a choice or sequence of two triple expressions, or a repeated triple expression:

```
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
```

The `extends` case is what breaks the 2017 proof pattern at its foundation. In the 2017 semantics, a shape reference is resolved purely by consulting the typing, so `Satisfies`/`Matches` are well-founded by ordinary structural recursion on the finite syntax tree. An `extends` clause instead recurses into *other labels' own definitions*, which implies a schema lookup, not a subterm of the current expression, so a plain structurally-recursive Lean definition is not just inconvenient here, it would be outright *false* if the schema's extension hierarchy happened to be cyclic. The paper handles this with its own Lemma 1 (termination, proved separately from the semantics), so the Lean development follows the same shape: `Satisfies` is first defined as a total, `Fuel`-indexed under-approximation. This is a standard technique for giving a structurally-terminating definition to a recursion whose termination is not otherwise obvious to Lean: an extra `Fuel : ℕ` argument is decremented on every recursive call, the recursion returns `False` outright once `Fuel` reaches `0`, and it is this decrease in `Fuel` — not the shape of the schema — that Lean accepts as structural:

```
def SatisfiesFuel : (Fuel : ℕ) → Schema → Graph → Typing → RDFNode →
    ShapeExpr → Prop
  | 0, _, _, _, _, _ ⇒ False
  ...
```

and a separate file, `Termination.lean`, proves that for a well-defined schema some large-enough, canonical fuel value makes the answer stop depending on `Fuel` at all, recovering the paper's real, fuel-free `Satisfies`. Turning the paper's one-sentence termination argument ("the tree of recursive calls is finite due to the acyclic nature of the extension hierarchy") into a Lean proof needs an actual rank function on labels witnessing that acyclicity, combined into a single measure that decreases on both kinds of recursive step: ordinary AST descent and `extends`-hops.

Definition 4 (correct typing) also gains a genuinely new case beyond a direct port of the 2017 statement: an abstract label is satisfied not by its own definition, but by the *existence* of some concrete descendant that is:

```
def Schema.CorrectTyping (S : Schema) (G : Graph) (τ : Typing) : Prop :=
  ∀ n z, τ.has n z →
    (z ∉ S.A → Satisfies S G τ n (S.shapeDef z)) ∧
    (z ∈ S.A → ∃ x ∈ S.desc z \ S.A, Satisfies S G τ n (S.shapeDef x))
```

From there, Proposition 1, Lemma 2, and Algorithm 1 are structurally direct ports of their 2017 counterparts — same per-stratum greatest-fixed-point idea for the algorithm, same Fusion/Swap/Independence argument for Lemma 2 (whose own paper proof, again, is only sketched as "Proof. (Idea)...") — but none of them transfer *mechanically*. The 2017 monotonicity and schema-agreement lemmas these all lean on go through by plain structural recursion, because 2017's `Satisfies` never dereferences the schema. Inheritance's `Satisfies` does, via ancestor lookups, so every one of those lemmas first had to be re-proved against the schema's dependency structure and then generalized from a single stratification-tied restriction (`stratRestrict`) to an arbitrary dependency-closed label set (`Schema.restrictLabels`), before Lemma 2's schedule-agnostic argument and Algorithm 1's per-stratum schema-swapping could reuse it. Algorithm 2 (*prove*) has not yet been ported to this formalism.

In order to formalize the definitions we needed to tackle some potential places in which the paper did not clearly specify all the conditions. For example, Table 3's `extra` rule is *not monotone* in the typing without a further condition: growing the typing can move a triple out of the "extra-covered" bucket and into the main triple-expression match, which can break a bare (non-starred) triple constraint that needed an exact singleton. For instance, let node n have two triples $t_1 = (n, p, o_1)$ and $t_2 = (n, p, o_2)$, and let h be a shape with `closed := false`, `extra := some {p}` and `triples := p @z`. Under $\tau_1 = \{(o_1, z)\}$, t_1 matches `p @z` and t_2 is covered by `extra`, so n satisfies h. Under $\tau_2 = \tau_1 \cup \{(o_2, z)\}$ both triples must match `p @z`, which accepts exactly one, so n no longer satisfies h although $\tau_1 \subseteq \tau_2$. The appendix proof of Proposition 1 in the paper states the fix for this: an invariant it calls "(*)": a shape's triple-constraint properties never overlap its own `extra` set. It is not stated as a standalone definition in the main text: we named it explicitly as `Schema.WellFormedExtra` and added it as a hypothesis to Proposition 1, Lemma 2 and Algorithm 1 of the inheritance formalism.

9 Conclusions and future work

Using Lean, it is possible to machine-check the different lemmas that relate the semantics of the ShEx language variants. The resulting GitHub repository [3] contains the ShEx schemas and the inheritance paper semantics and proofs, all the examples from both papers associated with their corresponding numbers, an executable main with two command-line options: `list` to show all the propositions and examples that can be checked, and `run` to check a specific one by its name. The development builds with Lean 4 (`v4.34.0-rc1`). It comprises about 12.7k lines of Lean (9.5k of definitions and proofs, 3.2k of worked examples) and about 420 theorems: 189 for the 2017 semantics, 230 for inheritance and 4 shared. The Lean proofs are complex and were defined with the help of Claude Code showing that the use of generative AI tools can be useful to help automate the proof construction process.

This development can serve as a first approach to explore more design decisions about the ShEx language. Future work includes a more extensive exploration of the design space for RDF validation languages including extending ShEx for RDF 1.2, or other language features [10] as well as developing a library of Lean tactics for Semantic Web languages that can be used to shorten the proofs and verify the correctness of other similar languages and algorithms.

Acknowledgments

This work has been partially funded by the project SHAKIGRA: Shaping Knowledge and Interoperable Graphs, code: NAC-ES-PUB-ASV-2025 PID2024-157010OBI00, from the Spanish Research Agency, the regional project with code SEK25-GRUGIC-24-089 and COST Action CA23147 GOBLIN - Global Network on Large-Scale, Cross-domain and Multilingual Open Knowledge Graphs.

Declaration of use of Generative AI

The authors declare that the content of this paper and the accompanying Lean code was developed with the assistance of Claude Code (Sonnet 5), ChatGPT and GitHub Copilot. The authors have reviewed and edited the content as needed and take full responsibility for the content of this publication.

References

1. Ahmetaj, S., Boneva, I., Hidders, J., Hose, K., Jakubowski, M., Gayo, J.E.L., Martens, W., Mogavero, F., Murlak, F., Okulmus, C., Polleres, A., Savkovic, O., Simkus, M., Tomaszuk, D.: Common foundations for SHACL, ShEx, and PG-Schema. In: The Web Conference (WWW). pp. 8–21. ACM (2025). https://doi.org/10.1145/3696410.3714694, https://doi.org/10.1145/3696410.3714694

[3] https://github.com/weso/shex_lean/tree/kgswc_paper

2. Ahmetaj, S., Boneva, I., Hidders, J., Jakubowski, M., Labra-Gayo, J.E., Martens, W., Mogavero, F., Murlak, F., Okulmus, C., Savković, O., Šimkus, M., Tomaszuk, D.: Common foundations for recursive shape languages. In: 23rd International Conference on Principles of Knowledge Representation and Reasoning, KR'26 (Jul 2026), https://arxiv.org/abs/2604.20946
3. 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) (2023). https://doi.org/10.1145/3589778, https://doi.org/10.1145/3589778
4. Boneva, I., Labra Gayo, J.E., Prud'hommeaux, E.G.: Semantics and validation of shapes schemas for RDF. In: International Semantic Web Conference (ISWC). pp. 104–120 (2017)
5. Boneva, I., Labra Gayo, J.E., Prud'hommeaux, E.G., Thornton, K., Waagmeester, A.: Shape expressions with inheritance. In: European Semantic Web Conference (ESWC). pp. 481–499 (2025)
6. Cutler, J.W., Disselkoen, C., Eline, A., He, S., Headley, K., Hicks, M., Hietala, K., Ioannidis, E., Kastner, J., Mamat, A., McAdams, D., McCutchen, M., Rungta, N., Torlak, E., Wells, A.M.: Cedar: A new language for expressive, fast, safe, and analyzable authorization. Proc. ACM Program. Lang. **8**(OOPSLA1) (2024). https://doi.org/10.1145/3649835
7. Huang, L., Ebersold, S., Kogtenkov, A., Meyer, B., Liu, Y.: Lessons from formally verified deployed software systems. ACM Computing Surveys **58**(8) (2026). https://doi.org/10.1145/3785652, extended version: https://arxiv.org/abs/2301.02206
8. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: ACM Symposium on Operating Systems Principles (SOSP). pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596
9. Knublauch, H., Kontokostas, D.: Shapes Constraint Language (SHACL). W3C Recommendation, W3C (2017), https://www.w3.org/TR/shacl/
10. Labra Gayo, J.E.: Extending shape expressions for different types of knowledge graphs. In: Data Quality meets Machine Learning and Knowledge Graphs (DQMLKG). vol. 3747. CEUR-WS.org, Aachen (2024), https://ceur-ws.org/Vol-3747/
11. Labra Gayo, J.E., Prud'hommeaux, E., Staworko, S., Solbrig, H.: Towards an RDF validation language based on regular expression derivatives. In: Proceedings of the International Workshop on Linked Web Data Management (LWDM 2015) (2015)
12. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
13. Leroy, X.: Mechanized semantics for compiler verification. In: Certified Programs and Proofs (CPP). Lecture Notes in Computer Science, vol. 7679, pp. 386–388. Springer (2012). https://doi.org/10.1007/978-3-642-35308-6_2
14. Moura, L.d., Ullrich, S.: The Lean 4 Theorem Prover and Programming Language, pp. 625–635. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-79876-5_37
15. Paulson, L.C., Nipkow, T., Wenzel, M.: From lcf to isabelle/hol. Formal Aspects of Computing **31**, 675–698 (2019). https://doi.org/10.1007/s00165-019-00492-1

16. Prud'hommeaux, E., Boneva, I., Labra Gayo, J.E., Kellogg, G.: Shape expressions language 2.1. W3C community group report, W3C (2019), http://shex.io/shex-semantics/
17. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.: Shape expressions: an RDF validation and transformation language. In: International Conference on Semantic Systems (SEM). pp. 32–40. ACM (2014). https://doi.org/10.1145/2660517.2660523, https://doi.org/10.1145/2660517.2660523
18. Staworko, S., Boneva, I., Labra Gayo, J.E., Hym, S., Prud'hommeaux, E., Solbrig, H.: Complexity and expressiveness of ShEx for RDF. In: International Conference on Database Theory (ICDT). pp. 195–211 (2015)
19. Tantow, J., Gerlach, L., Mennicke, S., Krötzsch, M.: Verifying Datalog Reasoning with Lean. In: Forster, Y., Keller, C. (eds.) 16th International Conference on Interactive Theorem Proving (ITP 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 352, pp. 36:1–36:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). https://doi.org/10.4230/LIPIcs.ITP.2025.36, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.36
20. The Rocq Development Team: The Rocq Reference Manual. INRIA (2026), https://rocq-prover.org/doc/V9.2.0/refman/, version 9.2.0
21. Turner, D., Carroll, J.J.: Comparing OWL Semantics. Tech. rep., HP Labs (2007), https://shiftleft.com/mirrors/www.hpl.hp.com/techreports/2007/HPL-2007-146.html

10 Appendix: Abstract syntax of ShEx (2017)

Fig. 1 of [4] defines a shapes schema as a pair (L, def), where L is a finite set of shape labels and def associates a shape expression with every label in L. A shape expression is generated by the following grammar:

```
ShExpr        ::= ValueDescr | NeigDescr | ShExpr AND ShExpr
                | ShExpr OR ShExpr | NOT ShExpr
ValueDescr    ::= a subset of IRI ∪ Lit ∪ {_b}
NeigDescr     ::= TExpr
TExpr         ::= TriplePattern | EMPTY
                | TExpr | TExpr | TExpr ; TExpr | TExpr [min;max]
TriplePattern ::= PropSet @ ShapeRef
PropSet       ::= a subset of IRI
ShapeRef      ::= a shape label in L
```

where `_b` is a constant standing for any blank node, `AND`, `OR` and `NOT` are the Boolean shape-expression operators, `|` (*SomeOfExpr*) and `;` (*EachOfExpr*) combine two triple expressions, and `[min;max]` (*RepetExpr*) repeats `TExpr` between `min` and `max` values, with `max = *` standing for unbounded.

This grammar corresponds directly to the Lean types introduced in Section 4: `ShExpr` encodes `ShExpr` (constructors `valueDescr`, `neigDescr`, `and`, `or`, `not`), and `TExpr` encodes `TExpr` (constructors `tPattern`, `empty`, `someOf`, `eachOf`, `rep`). `PropSet` is represented as `PropSet`, a `Finset IRI`. `ValueDescr` is represented as `ValueDescr`, a `Finset ValueItem`, where `ValueItem` is itself an `inductive` type with one constructor per alternative of IRI (`iri`), literal (`lit`) or blank node (`anyBlank`). `ShapeRef` is simply a `Label` (an abbreviation for `String`): a triple pattern's shape reference is not resolved to a shape expression syntactically, only later, through a typing (Section 5).

11 Appendix: Abstract syntax of ShEx with inheritance

Definition 1 of [5] fixes two disjoint finite sets of shape labels, Y (*simple* labels) and E (*extendable* labels). A shape expression over Y and E is either a reference `@z`, or a plain shape expression derivable from the non-terminal `u` in the following grammar:

```
u ::= h | t | c | u or u | u and u | not u
h ::= [closed] [extra P] <e>
t ::= extends X h
c ::= a boolean function over IRI ∪ Blank ∪ Literal
e ::= ε | p @z | e|e | e;e | e*
```

where `u` is a plain shape expression, `h` a shape, `t` a shape with extends, `c` a node constraint, and `e` a triple expression; `z` $\in Y \cup E$, `X` $\subseteq E$, `p` $\in IRI$, `P` $\subseteq IRI$, the symbol `ε` is a constant, and `@`, `|`, `*` and `;` are operators.

This grammar corresponds directly to the Lean types introduced in Section 8: `TripleExpr` encodes `e` (constructors `empty`, `tc`, `choice`, `seq`, `star`), and `ShapeExpr` encodes `u` (constructors `ofShape`, `extends`, `constraint`, `or`, `and`, `not`). The `@z` reference is not modeled as its own constructor: it only ever occurs nested inside a triple constraint `p @z`, so it is represented directly by the `tc` constructor of `TripleExpr` rather than by a recursive case of `ShapeExpr`.