



# Verifying Datalog Reasoning with Lean

Johannes Tantow  




TU Chemnitz, Germany

Lukas Gerlach   

Knowledge-Based Systems Group, TU Dresden, Germany

Stephan Mennicke   

Knowledge-Based Systems Group, TU Dresden, Germany

Markus Krötzsch   

Knowledge-Based Systems Group, TU Dresden, Germany

---

## Abstract

Datalog is an essential logical rule language with many applications, and modern rule engines compute logical consequences for Datalog with high performance and scalability. While Datalog is rather simple and, in principle, explainable by design, such sophisticated implementations and optimizations are hard to verify. We therefore propose a certificate-based approach to validate results of Datalog reasoners in a formally verified checker for Datalog proofs. Using the proof assistant Lean, we implement such a checker and verify its correctness against direct formalizations of the Datalog semantics. We propose two JSON encodings for Datalog proofs: one using the widely supported Datalog proof trees, and one using directed acyclic graphs for succinctness. To evaluate the practical feasibility and performance of our approach, we validate proofs that we obtain by converting derivation traces of an existing Datalog reasoner into our tool-independent format.

**2012 ACM Subject Classification** Software and its engineering → Formal software verification; Computing methodologies → Theorem proving algorithms; Theory of computation → Constraint and logic programming; Theory of computation → Automated reasoning

**Keywords and phrases** Certifying Algorithms, Datalog, Formal Verification

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2025.36

**Supplementary Material** *Software (Lean Formalization)*: [github:knowsys/CertifyingDatalog](https://github.com:knowsys/CertifyingDatalog)

**Funding** This work was partly supported by DFG (German Research Foundation) in project 389792660 (TRR 248, Center for Perspicuous Systems) and in the CeTI Cluster of Excellence; by the Bundesministerium für Bildung und Forschung (BMBF) in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI) and in project 13GW0552B (KIMEDS); by BMBF and DAAD (German Academic Exchange Service) in project 57616814 (SECAI, School of Embedded and Composite AI); and by the Center for Advancing Electronics Dresden (cfaed).

**Acknowledgements** We would like to thank Markus Himmel for his help and useful feedback during the integration of our results on hash maps into the standard library.

## 1 Introduction

Datalog is a simple and elegant rule language that is at the heart of many approaches to logic programming, knowledge representation, and data analysis [1, 14]. Syntactically, Datalog rules are universally quantified, function-free predicate logic implications, such as the following two recursive rules defining transitivity (written from right to left and without the implicit universal quantifiers, as is customary in logic programming):

$$\text{trans}(x, y) \leftarrow \text{edge}(x, y) \tag{1}$$

$$\text{trans}(x, z) \leftarrow \text{trans}(x, y) \wedge \text{trans}(y, z) \tag{2}$$



© Johannes Tantow, Lukas Gerlach, Stephan Mennicke, and Markus Krötzsch;  
licensed under Creative Commons License CC-BY 4.0

16th International Conference on Interactive Theorem Proving (ITP 2025).

Editors: Yannick Forster and Chantal Keller; Article No. 36; pp. 36:1–36:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The main reasoning task then is to compute all logical consequences of such *programs* over a given set of facts (e.g., for *edge*), which can equivalently be seen as first-order entailment or second-order model checking [1].<sup>1</sup> Deciding such consequences is P-complete with respect to the set of input facts, making Datalog interesting for data-intensive applications.

Datalog is well supported in practice. Modern rule engines such as *clingo* [19], *Nemo* [22], *RDFox* [31], *Soufflé* [23], and *VLog* [34] can easily compute millions of Datalog consequences on commodity hardware. Tools like *Datomic* (<https://datomic.com/>) and Google’s *Logica* (<https://logica.dev/>) are database-oriented and leverage the robustness of existing data management infrastructure. In each case, practical tools include many additional features and language extensions and rely on intricate optimizations to achieve the necessary performance.

Unfortunately, the complexity of real-world systems unavoidably leads to errors that may compromise correctness. Mature systems like the above counteract this with extensive testing, code quality analysis, and public issue trackers, but complete freedom of bugs is rare and volatile in such ever-evolving systems [26, 27, 37]. Stronger correctness guarantees are given by certified Datalog systems, but the few prototypes that exist cannot scale to the input sizes of optimized systems yet [7], or even restrict to a subset of Datalog [8].

In this paper, we address this challenge by developing a certificate checker for Datalog, written and verified in Lean 4[30]. Instead of replacing optimized systems, such checkers validate certificates that prove the correctness of individual results. Successful uses of this approach (a.k.a. the *de Bruijn* criterion [6]) exist in various fields [21, 5, 18, 35, 9], but we are not aware of any solution for Datalog. This is surprising since the logical semantics of Datalog leads to structured proofs that are suitable certificates. Indeed, systems like *Nemo* and *Soufflé* include tracing features that can produce such *proof trees* for their derivations.

Our main concerns are correctness and performance. For correctness, we formalize the semantics of Datalog in two ways – least models and proofs – and prove their equivalence. We then formalize a checker that builds upon the proof-theoretic semantics to validate given proofs. This also requires us to formalize Datalog syntax, unification, and model theory.

Towards practical performance, we allow the checker to ingest JSON-formatted inputs that encode Datalog rules and (sets of) proofs in an implementation-independent exchange format. For proofs, we consider a traditional encoding in the form of *proof trees* and a redundancy-avoiding encoding as *proof graphs* that can be exponentially more succinct. We evaluate the feasibility and performance of our approach using synthetic and real Datalog tasks and proofs generated by the Datalog engine *Nemo*.

This paper hyperlinks most Lean-symbols to their formal definitions in our repository. Some proofs are abbreviated in the code blocks by `_`, but are available in the linked code. All proofs in this paper reflect the ideas of the formal Lean proofs in natural language.

## 2 Formalizing Datalog

We begin by formalizing Datalog and two definitions of its semantics, which we then show to be equivalent. Datalog programs are based on a *Signature* that provides sets of *constants*, *variables*, and *relation symbols*, where each relation has an arity  $\geq 0$ :

<sup>1</sup> The views do differ for other reasoning tasks, notably program entailment (a.k.a. query containment), which is decidable for first-order rules but undecidable for second-order programs.

```

structure Signature where
  (constants: Type)
  (vars: Type)
  (relationSymbols: Type)
  (relationArity: relationSymbols → ℕ)

```

If not stated otherwise, we use a fixed **Signature**  $\tau$ . A **Term** is a constant or a variable; our formalization uses distinct constructors for each case so that constants and variables are disjoint. Further restrictions are not required in our work.

An **Atom** is an expression of the form  $p(t_1, \dots, t_n)$ , where  $t_1, \dots, t_n$  are terms and  $p$  is a relation of arity  $n$ . A **Rule** has the form  $H \leftarrow B_1 \wedge \dots \wedge B_n$ , where the *head*  $H$  is an **Atom**, and the *body*  $B_1 \wedge \dots \wedge B_n$  is a conjunction of **Atoms** (represented as a list in Lean). A **Program** is a finite set of rules. A **GroundAtom** (or *fact*) is an **Atom** without variables and a **Database** is a finite set of facts. A **KnowledgeBase** is a pair of a **Program** and a **Database**.

The semantics of Datalog is given by defining the set of all facts that are logical consequences of a knowledge base. A (possibly infinite) set of facts is an **Interpretation**, which can indeed be seen as a first-order *Herbrand* interpretation, interpreting constant symbols by themselves (a.k.a. *unique name assumption*). Several equivalent definitions for the semantics of Datalog exist, e.g., based on least models, proof trees, and fixed-point operators [1]. We formalize the first two – model-theoretic and proof-theoretic – semantics, and validate their correctness by proving the equivalence of these distinct definitions.

Both semantics consider **GroundRules**, i.e., rules that contain only **GroundAtoms**, obtained from rules by mapping variables to constants. Such a mapping  $g$  is a **Grounding**, and its application to a rule is formalized as **applyRule**. For a **Program**  $P$ , the **groundProgram**  $\text{ground}(P)$  is the set of all ground rules  $r$  obtained from some  $r' \in P$  by some grounding  $g$ :

```

def Program.groundProgram (p : Program  $\tau$ ) := {r : GroundRule  $\tau$  |  $\exists$  (r' : Rule  $\tau$ )
  (g : Grounding  $\tau$ ), r'  $\in$  p  $\wedge$  r = g.applyRule' r'}

```

An interpretation  $I$  satisfies a ground rule  $r$  if, whenever  $I$  contains all body atoms of  $r$ ,  $I$  also contains the head of  $r$ .  $I$  satisfies a **Program**  $P$  if  $I$  satisfies all rules  $r \in \text{ground}(P)$ . Finally,  $I$  **models** a knowledge base  $\langle P, D \rangle$  if  $I$  satisfies  $P$  and  $D \subseteq I$ . The **modelTheoreticSemantics** for  $\langle P, D \rangle$  is the least model of  $\langle P, D \rangle$ :

```

def modelTheoreticSemantics (kb : KnowledgeBase  $\tau$ ) : Interpretation  $\tau$  :=
  {a : GroundAtom  $\tau$  |  $\forall$  (i : Interpretation  $\tau$ ), i.models kb  $\rightarrow$  a  $\in$  i}

```

The **proofTheoreticSemantics** of a knowledge base  $\langle P, D \rangle$  is the set of all facts that have a **ProofTree** valid in  $P$  and  $D$ . Listing 1 shows our formalization: a directed node-labeled tree  $\mathcal{T}$  of ground atoms is a **ProofTreeSkeleton** and it is *valid* in  $P$  and  $D$  for a ground atom  $a$  if (1) the root of  $\mathcal{T}$  is labeled by  $a$  and (2) for all nodes  $t$  (including the root) either (2a)  $t$  is a leaf and the label of  $t$  occurs in  $D$ , or (2b) if  $s_1, \dots, s_k$  are the children of  $t$  in  $\mathcal{T}$ , then there is a rule  $r \in \text{ground}(P)$  such that the head of  $r$  is  $t$  and the body atoms of  $r$  are  $s_1, \dots, s_k$ . A **ProofTree** is then a valid proof tree skeleton. We can now define:

```

def proofTheoreticSemantics (kb : KnowledgeBase  $\tau$ ) : Interpretation  $\tau$  :=
  {a : GroundAtom  $\tau$  |  $\exists$  (t : ProofTree kb), t.root = a}

```

In **modelAndProofTreeSemanticsEquivalent**, we formally verify the equivalence of the semantics. The proof uses the fact that the **modelTheoreticSemantics** yields the least model together with the subsequent lemmas.

```

inductive Tree (A: Type u)
| node: A → List (Tree A) → Tree A

abbrev ProofTreeSkeleton (τ: Signature) := Tree (GroundAtom τ)

def ProofTreeSkeleton.isValid (P: Program τ) (kb : KnowledgeBase τ)
(t: ProofTreeSkeleton τ): Prop :=
  match t with
  | .node a l =>
    (∃ (r: Rule τ) (g: Grounding τ), r ∈ kb.prog
      ∧ g.applyRule' r = {head:= a, body:= l.map root}
      ∧ l.attach.Forall (fun ⟨st, _h⟩ => isValid st kb))
    ∨ (l = [] ∧ kb.db.contains a)

```

■ **Listing 1** Proof tree skeletons and their validity

► **Lemma 1** (`proofTheoreticSemanticsIsModel`). *The `proofTheoreticSemantics` for a knowledge base  $\langle P, D \rangle$  is a model of  $\langle P, D \rangle$ .*

**Proof.** We show that the `proofTheoreticSemantics`  $F$  for  $\langle P, D \rangle$  contains  $D$  and satisfies  $\text{ground}(P)$ . The former is easy to show since every fact  $a$  in  $D$  has a valid proof tree that consists of a single node labeled with  $a$  (see `dbElementsHaveProofTrees`).

It remains to show that  $F$  satisfies every rule in  $\text{ground}(P)$ . If the body atoms  $B_1, \dots, B_n$  of a rule  $r \in \text{ground}(P)$  are in  $F$ , then they have by the definition of  $F$  proof trees  $\mathcal{T}_1, \dots, \mathcal{T}_n$  that are valid in  $P$  and  $D$ . Hence, we can construct a valid proof tree for the head  $H$  of  $r$  by introducing a root node that is labeled with  $H$  and that has  $\mathcal{T}_1, \dots, \mathcal{T}_n$  as children. ◀

► **Lemma 2** (`proofTreeAtomsInEveryModel`). *The `proofTheoreticSemantics` for a knowledge base  $\langle P, D \rangle$  is a subset of every model of  $\langle P, D \rangle$ .*

**Proof.** Let  $M$  be a model of  $\langle P, D \rangle$ . We show that  $M$  contains all facts  $a$  that are the root label of some proof tree  $\mathcal{T}$  that is valid in  $P$  and  $D$ . We show this by strong induction on the height  $h$  of  $\mathcal{T}$ . For the inductive step, assume the claim has been shown for all trees of height  $< h$ . There are two cases where  $\mathcal{T}$  is valid. In the first case,  $\mathcal{T}$  is just a single node that is in  $D$ . Hence, we have  $a \in M$ . In the second case, there is a rule  $r \in \text{ground}(P)$  with head  $a$ , such that the children  $s_1, \dots, s_k$  of the root of  $\mathcal{T}$  are labeled with the body atoms of  $r$ . Since the subtrees of  $\mathcal{T}$  with roots  $s_1, \dots, s_k$  have height  $< h$ , we can apply the induction hypothesis to conclude that each body atom of  $r$  is in  $M$ . But then the head of  $r$  (i.e.,  $a$ ) also occurs in  $M$  since  $M$  is a model. ◀

### 3 Implementing a Computable Check for Proof Tree Validity

In this section, we describe our implementation for a checker of proof tree validity. In contrast to the general formalizations in Section 2, this leads to an effective approach for verifying the soundness of individual conclusions, which can be applied on proof trees that are provided by existing Datalog implementations.

We implement a computable function `checkValidity` whose result coincides with the (uncomputable) `ProofTreeSkeleton.isValid`, shown in `checkValidityOkIffIsValid`.

### 3.1 Finding Substitutions with Unification

The formalizations in Section 2 use `Groundings` that are total functions from variables to constants. For an effective validity check, we take a similar approach as Benzaken et al. [7] and use partial functions (i.e., `Substitutions`) since they are easier to expand recursively.

```
def Grounding (τ: Signature) := τ.vars → τ.constants
def Substitution (τ: Signature) := τ.vars → Option (τ.constants)
```

The application of a substitution to a rule, `applyRule`, is defined in the obvious way. We verify that `Groundings` and `Substitutions` produce the same ground instances from any rule (`grounding_substitution_equiv`). For turning a substitution into a grounding, we just need some default constant to use for unmapped variables.

```
theorem grounding_substitution_equiv [Inhabited τ.constants]:
  (∃ (g: Grounding τ), g.applyRule' r' = r) ↔
  (∃ (s: Substitution τ), s.applyRule r' = r)
```

To validate a proof tree, we need to check that the label of every non-leaf node can be entailed by applying a ground rule to the labels of its children. Since the set of ground rules can be large, even infinite, a goal-directed procedure is needed. We therefore implement a *unification* algorithm that finds a `Substitution` to map a given `Rule` to a given `GroundRule`, if such a `Substitution` exists. The main outcome is the function `checkRuleMatch` together with a proof of its correctness (`checkRuleMatchOkIffExistsRule`).

```
def matchRule (r: Rule τ) (gr: GroundRule τ): Option (Substitution τ) :=
  ((empty.matchAtom r.head gr.head).bind fun s => s.matchAtomList (r.body.zip
    gr.body)).filter (fun _ => r.body.length = gr.body.length)
```

The function `matchRule` first matches the head atoms and then all body atoms of the rules, each time updating the current `Substitution`. Atoms are mapped by comparing their relation symbols and iterating over their terms, and applying the function `matchTerm` to each corresponding pair. That function either returns a (possibly extended) `Substitution`, or `Option.none` if a previously mapped variable is to be unified with a different constant.

```
def matchTerm (t: Term τ) (c: τ.constants) (s: Substitution τ):
  Option (Substitution τ) :=
  match t with
  | .constant c' => if c = c' then Option.some s else Option.none
  | .variableDL v =>
    (some (extend s v c)).filter (fun s' => (s v).isSome → s v = s' v)
```

This definition is accompanied by lemmas showing that `matchTerm` returns a substitution if and only if one exists (`matchTermYieldsSubs`, `matchTermNoneThenNoSubs`) and that the returned substitution is subset-minimal with respect to its `domain` (`matchTermIsMinimal`).

### 3.2 Validation for Single Trees

A proof tree can be validated for  $\langle P, D \rangle$  by checking that each of its leaf nodes occurs in  $D$  or in a ground rule with an empty body in  $P$ , and each of its non-leaf nodes represents a valid inference using a rule in  $P$ . For the latter, we can directly read off the required ground rule from any inner node, and it remains to check if it can be obtained from a rule in  $P$ . Some Datalog tools generate proof trees that specify the original rule in  $P$  for each step, but it is also possible that only its ground instance is provided. We therefore design our

```

def checkValidity (t : ProofTreeSkeleton  $\tau$ ) (m : SymbolSequenceMap  $\tau$ )
  (d : Database  $\tau$ ) : Except String Unit :=
  match t with
  | .node a l =>
    if l.isEmpty
    then if d.contains a
         then Except.ok ()
         else
           (checkRuleMatch m {head:= a, body := []}).map (fun _ => ())
    else
      (checkRuleMatch m {head:= a, body := l.map Tree.root}).bind (fun _ =>
        (l.attach.mapExceptUnit (fun (t, _h) => checkValidity t m d)))

```

■ Listing 2 Goal-directed validation of proof trees

implementation so that it finds a suitable rule from  $P$  even if not given explicitly. However, we do assume that the leaves in the proof tree are provided in the order of atoms in the original rule, which is natural and indeed true for all implementations we are aware of.

To find applicable rules efficiently, we associate each rule with the list of relation symbols in its head and body atoms, in their order of appearance:

```

def symbolSequence (r: Rule  $\tau$ ) := r.head.symbol :: (List.map Atom.symbol r.body)

```

We then construct a lookup structure  $m$ : `Std.HashMap (List  $\tau$ .relationSymbols) (List (Rule  $\tau$ ))` mapping lists of relation symbols to the list of all corresponding rules in  $P$ . This allows us to find rules for a given ground rule  $r$  in `checkRuleMatch` and to prove that the check returns `Option.ok ()` if and only if  $r$  is in  $\text{ground}(P)$  (`checkRuleMatchOkIffExistsRule`):

```

def checkRuleMatch (m: SymbolSequenceMap  $\tau$ )
  (gr: GroundRule  $\tau$ ) : Except String Unit :=
  if (m.find gr.toRule.symbolSequence).any (fun rule => (Substitution.matchRule
    rule gr).isSome)
  then Except.ok ()
  else Except.error ("No match for " ++ ToString.toString gr)

```

Listing 2 shows the overall function `checkValidity` as our practical implementation to verify `ProofTreeSkeleton.isValid`. The straightforward check that leaf nodes of the proof tree are in the database is also included. As an alternative, we also allow for facts being the conclusion of rules with empty bodies. For inner nodes we only have to check whether this forms a ground rule of the program. We show that this implementation does indeed agree with the definition in `ProofTreeSkeleton.isValid`, which is established by induction on the height of the input tree (`checkValidityOkIffIsValid`).

## 4 From Proof Trees to Proof Graphs

Trees are a traditional representation of proofs in logic programming, and many practical tools provide tracing features in terms of tree structures. However, this is far from optimal, as the following example illustrates.

► **Example 3.** Consider the following rules with variables  $x$ ,  $y$  and  $z$ :

$$\text{trans}(x, y) \leftarrow \text{edge}(x, y) \quad (3) \qquad \text{t}(x, y) \leftarrow \text{trans}(x, y) \quad (5)$$

$$\text{trans}(x, z) \leftarrow \text{t}(x, y) \wedge \text{u}(x, y) \wedge \text{edge}(y, z) \quad (4) \qquad \text{u}(x, y) \leftarrow \text{trans}(x, y) \quad (6)$$

```

variable {A: Type} [DecidableEq A] [Hashable A]
variable (A) in abbrev PreGraph := Std.HashMap A (List A)
def PreGraph.vertices (g : PreGraph A) : List A := g.toList.map Prod.fst
def PreGraph.predecessors (g : PreGraph A) (a : A) : List A := g.findD a []
def PreGraph.complete (g:PreGraph A):= ∀ a ∈ g, (g.predecessors a).all fun x => x ∈
  g
variable (A) in abbrev Graph := { pg : PreGraph A // pg.complete }

```

■ **Listing 3** The hashmap based graph model

This still defines `trans` as the transitive closure of `edge`. Given a chain `edge(a0, a1), ..., edge(an, an+1)`, we can derive `trans(a0, an+1)` in a proof that uses rule (4)  $n$  times. However, since (4) has body atoms for `t` and `u`, the proof of `trans(a0, an+1)` contains two disjoint subtrees with the proof of `trans(a0, an)`, i.e., it is exponential in  $n$ . To avoid this, we would rather consider only a single proof of `trans(a0, an)`, which can then be used by rules (5) and (6) to show `t(a0, an)` and `u(a0, an)`, respectively.

We therefore propose the use of directed acyclic graphs to allow more efficient encodings of certificates. The special case of proof trees is still supported, but optimized outputs are possible if a tool supports them. Lean has graph-theoretic primitives and theorems in its *mathlib*, but their formalization was not sufficient for our purposes. First, *mathlib* currently considers undirected graphs and has only very limited results on directed graphs which is crucial to model our problem. Second, its graph model uses an adjacency matrix approach, but we expect the proof graphs to be sparse and therefore expect an adjacency list. Instead, we design two new graph encodings – referred to as *unordered* and *ordered* –, which we implement using `HashMap` and `Array`, respectively. Both encodings resemble adjacency lists in the sense that each atom is directly associated with all atoms that are used for its derivation.

Both approaches require that the graph is acyclic in order to capture a valid derivation, but differ in the way of achieving this requirement. The ordered approach requires a topological sorting of the graph already as the input and therefore allows faster checking and a more compact representation. The output of reasoners does however not reflect this input format and is instead often just a list of edges. These can easily be verified using the unordered graph approach which computes the topological sorting.

## 4.1 Modeling a Directed Acyclic Graph

Our underlying graph model, called `PreGraph`, is a `HashMap` from Lean’s standard library mapping each vertex to a list of its immediate predecessors. Alternatively, we could have used a simple function from vertices to their predecessors, but this was too slow in practice. Hashmaps were faster but required us to prove additional properties of certain operations. These results have been contributed to the standard library as a byproduct of this work.

Now a `Graph` is a `PreGraph` that is *complete* in the sense that each vertex that occurs as a predecessor also occurs as a key in the map (possibly with an empty list of predecessors). This prevents ill-defined graphs where a vertex is connected via an edge to something that is not a vertex in our graph. An alternative would be to have a special type that consists of all vertices, but this approach was easier for the parsing.

A *walk* is a list of connected vertices. We use lists and place the first vertex at the end of the list since Lean’s inductive list definition makes it easier to prepend new elements to lists. This choice influences the way we explore graphs later on. Formally, a possibly empty



list of vertices  $\ell = v_0, \dots, v_n$  `isWalk` in a graph  $G$  if all vertices in  $\ell$  occur in  $G$  and, for all  $0 < i \leq n$ ,  $v_{i-1}$  is a predecessor of  $v_i$  in  $G$ :

```
def List.isWalk (l : List A) (G: Graph A) : Prop :=
  (∀ (a:A), a ∈ l → a ∈ G.vertices) ∧
  ∀ i > 0, ∀ (g: i < l.length), l[i.pred]’_ ∈ G.predecessors l[i]
```

Using lists is convenient to extend walks by prepending vertices:

► **Lemma 4** (`prependPredecessor`). *Let  $w$  be a walk in a graph  $G$  of the form  $a :: tl$ , and let  $b$  be a predecessor of  $a$ . Then  $b :: a :: tl$  is a walk in  $G$ .*

A graph that encodes a well-founded proof should also be acyclic. Formally, a walk `isCycle` if it consists of at least two vertices and has the same vertex in its first and last position. A graph `isAcyclic` if it does not have cycles.

## 4.2 Validating a Proof Graph via Depth-First Search

Recall that we are considering a knowledge base  $\langle P, D \rangle$  but instead of a proof tree, we now consider a graph  $G$  of facts. We verify that each vertex in  $G$  is the head of a rule  $r \in \text{ground}(P)$  such that the predecessors are exactly the body of  $r$  or that the vertex has no predecessors and, therefore, represents a database element. We abstract this into a `NodeCondition` as a function of the type  $A \rightarrow \text{Except String Unit}$ . Note that the graph stays constant during the exploration so that we can express our criteria in this form.

In general, we need a procedure that checks a node condition on all vertices and that also ensures that the graph is acyclic. We consider a standard depth-first search implementation [15] alongside some helper functions that ease our proof efforts. The algorithm is divided into two functions: (1) `verify_via_dfs` initializes the search on all vertices and (2) `verify_via_dfs_step` explores the graph recursively until no new vertex can be discovered. For early termination, we keep track of a set of vertices that have already been visited somewhere in the search. Therefore, the function `verify_via_dfs_step` also returns a set of all vertices that have been visited so far. For performance reasons, this set of vertices is again stored in a `HashSet`.

The function `verify_via_dfs` takes only the graph and a node condition, for which `verify_via_dfs_step` (defined below) is called on every vertex.

```
def verify_via_dfs (G : Graph A) (cond : NodeCondition A) : Except String Unit :=
  (G.vertices.attach.foldl_except
    (fun acc (a, h) => G.verify_via_dfs_step a cond (Walk.singleton G a h, _) acc)
    (Except.ok HashSet.empty)).map (fun _ => ())
```

The function `verify_via_dfs_step` (Listing 4) takes five arguments: (1) the current vertex  $a$ ; (2) the graph  $G$ ; (3) a node condition  $cond$  to be evaluated on  $a$  (4) the path taken up until  $a$ ; and (5) a set of already visited (i.e., verified) vertices. The function returns either a set of already verified vertices or an error (if the acyclicity check or  $cond$  fails).

Using the `verifiedNodes` set, we make sure to never explore a node twice. When exploring a node we check for the node condition and if any predecessor occurs already in the walk as this would indicate a cycle. If neither of these checks throws an error, we recursively explore the predecessors. Here, `foldl_except` is just used for aggregating the results of all branches (i.e., predecessors) of  $a$ . If any branch yields an error, the aggregation returns the same error. After each successful recursive call, the set `verifiedNodes` is extended by the explored vertices for the next call. Take note of the difference between and the different purposes of `walkFromA` and `verifiedNodes`. The former grows along the path we take in the depth-first



```

def verify_via_dfs_step (a : A) (G : Graph A) (cond : NodeCondition A)
  (walkFromA : {w : Walk G // w.val.head? = some a}) (verifiedNodes : HashSet
A) : Except String (HashSet A) :=
  if verifiedNodes.contains a then Except.ok verifiedNodes
  else (cond a).bind (fun _ =>
    if pred_not_mem_walk : (G.predecessors a).any
      (fun pred => pred ∈ walkFromA.val.val)
    then Except.error "Cycle detected"
    else
      let verifiedAfterRecursion :=
        (G.predecessors a).attach.foldl_except (fun verified ⟨pred, mem⟩ =>
          let walkFromPred : {w : Walk G // w.val.head? = some pred} :=
            ⟨walkFromA.val.prependPredecessor pred _, _⟩
          G.verify_via_dfs_step pred cond walkFromPred verified)
        (Except.ok verifiedNodes)
      verifiedAfterRecursion.map (fun verified => verified.insert a))

```

■ **Listing 4** Function to perform a single step in depth-first search

search (“top-down”) and is used to detect cycles whereas the latter grows while backtracking (“bottom-up”) and is used to mark vertices as fully verified.

The desired main correctness result of the procedure is formalized in the following theorem, which we prove in Section 4.3.

► **Theorem 5.** *Let  $G$  be a graph and  $cond$  be a node condition. Then  $dfs\ G\ cond = Except.ok\ ()$  if and only if  $G$  is acyclic and  $cond$  holds for all vertices in  $G$ .*

First, we complete the verification of the “proof graph” by plugging in the correct function for  $cond$ .

```

def locallyValid_for_kb (G : Graph (GroundAtom τ))
  (kb : KnowledgeBase τ) (node : GroundAtom τ) : Prop :=
  (∃ r ∈ kb.prog, ∃ (g : Grounding τ), g.applyRule' r =
    { head := node, body := G.predecessors node })
  ∨ (G.predecessors node = [] ∧ kb.db.contains node)

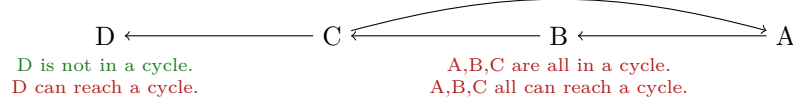
```

We say, a vertex  $v$  in a graph is `locallyValid_for_kb` for a knowledge base  $\langle P, D \rangle$  if  $v$  follows from its predecessors via a ground rule in  $ground(P)$  or  $v$  occurs in  $D$ .

If we have an acyclic graph with only locally valid vertices, proof trees for every vertex can be extracted. Hence, every fact encoded in such a graph is part of the proof-theoretic semantics (`verticesOfLocallyValidAcyclicGraphAreInProofTheoreticSemantics`). We computably check `locallyValid_for_kb` in `checkValidity` reusing `checkRuleMatch` similar to `checkValidity`. Correctness is proven in `checkValidityIsOkIffAcyclicAndAllValid`.

### 4.3 Correctness of Depth-First Search

We illustrate the proof idea for Theorem 5 used in our Lean implementation. Recall that `verify_via_dfs` calls `verify_via_dfs_step` on every vertex of the graph. In contrast to this procedure, acyclicity was defined by the absence of cycles. In order to prove the correctness of the implementation, we need a criterion that uses vertices explicitly. Then we can show that `verify_via_dfs_step` returns `ok` if and only if this criterion holds for the given vertex.



■ **Figure 1** Propagating Acyclicity Check Results via Depth-First Search in Example 6.

A candidate for this is *membership in a cycle*. A graph is acyclic if and only if every vertex is not a member of a cycle. This criterion, however, is insufficient for our implementation as the subsequent example shows.

► **Example 6.** Consider the graph depicted in Figure 1. We can start a depth-first search from vertex  $D$  following the predecessor relation. In the process, we want to check if the graph is acyclic by maintaining a list of vertices that corresponds to the path we have taken in the graph and checking if the next vertex already occurs in this list. In the graph, after visiting  $D, C, B, A$  (in that order), we visit  $C$  again, which is already in the list. We now know that  $C$  occurs in a cycle and also that  $A$  and  $B$  occur in this cycle. However,  $D$  does not occur in any cycle. Hence, if we pick membership in a cycle as our property for cycle detection, we could not faithfully propagate this information back to  $D$ . This means that `verify_via_dfs_step` should return an error while  $D$  does not fulfill the *membership in cycle* criterion. Instead, if we are only interested in *reachability from a cycle*, then this property holds for all vertices  $A, B, C, D$  and can safely be backpropagated once we find the cycle through  $C$ .

The observation from Example 6 is generalized as follows: A graph is cyclic if and only if some of its vertices *are reached from* a cycle. A vertex *is reached from* a cycle if it can be reached from a vertex that is part of a cyclic walk. We formalize this idea as follows: First, vertex  $a$  `canReach` vertex  $b$  if there is a non-empty walk from  $a$  to  $b$  (i.e.,  $b$  occurs in the reflexive and transitive closure of the predecessor relation of  $a$ ). Note that every vertex  $n$  can reach itself by  $[n]$ . Second, we formalize when a node is `reachableFromCycle` in graph  $G$ .

```
def canReach (G : Graph A) (a b : A) : Prop :=
  ∃ (w : Walk G) (neq: w.val ≠ []), (w.val.head neq) = a ∧ (w.val.getLast neq) = b

def reachableFromCycle (G: Graph A) (b : A) :=
  ∃ (w : Walk G), w.isCycle ∧ ∃ (a : A), a ∈ w.val ∧ G.canReach a b
```

The following lemma is an immediate consequence of the definitions.

► **Lemma 7** (`acyclicIffAllNotReachableFromCycle`). *A graph  $G$  is acyclic if and only if all vertices of  $G$  are not reached from a cycle.*

Furthermore, as illustrated in Example 6, we can propagate the property of a vertex being reached from a cycle to other vertices in a depth-first search.

► **Lemma 8** (`notReachableFromCycleIffPredecessorsNotReachableFromCycle`). *A vertex  $a$  is not reached from a cycle if and only if all predecessors  $b$  are not reached from a cycle.*

We reuse the property `canReach` for the following theorem for `verify_via_dfs_step`:

► **Theorem 9** (`dfs_semantics`). *In the scope of an application of `verify_via_dfs`, we say that a vertex  $n$  has property `DfsStepSemantics` if and only if  $n$  is not reached from a cycle (in  $G$ ) and if the node condition holds for every vertex  $m$  that reaches  $n$ .*

Consider the arguments of `verify_via_dfs_step` and require additionally that each vertex  $a'$  in `verifiedNodes` has the property *DfsStepSemantics*. Then, `verify_via_dfs_step` is successful if and only if  $a$  has the property *DfsStepSemantics*.

We will prove this theorem and multiple lemmas by induction on the cardinality of the finite set that contains all vertices that are not in the current walk for any input vertex, walk and set so that all elements of the input set fulfill the *DfsStepSemantics* property. Due to the input parameters of `verify_via_dfs_step` the base case is trivial since the current vertex is not in the path, but all vertices are in the path at the same time, which is a contradiction. Therefore, we only consider the induction step in the following.

The most challenging case is if we reach the `foldl_except` call. This function reduces the list one by one and produces new sets and reuses them in the next call. We can however break this into separate calls if some conditions hold. We note that the `verifiedNodes` set plays no role in the statement of Theorem 9 as long as all of its members have the *DfsStepSemantics* property. If we manage to prove that `verify_via_dfs_step` preserves the *DfsStepSemantics* property, we can replace the `foldl_except` call with individual calls of `verify_via_dfs_step` in the proof and directly use the induction hypothesis.

► **Lemma 10** (`dfs_step_result_valid`). *If the call of `verify_via_dfs_step` is successfully returning `verifiedAfter` and all elements of `verifiedNodes` had property *DfsStepSemantics*, then all elements of `verifiedAfter` have property *DfsStepSemantics*.*

**Proof.** The interesting case appears when  $a$  was not yet explored, as then the result is different from the input set. Since `verify_via_dfs_step` does not yield an error, the node condition must hold for  $a$ . The returned result `verifiedAfter` consists of  $a$  and the result of `foldl_except`. By inducing on the size of the list, we can prove that `foldl_except` preserves set properties if the function preserves it. This holds by the induction hypothesis so that all elements in `verifiedAfter' \ {a}` have the *DfsStepSemantics* property. Therefore it only remains to prove that  $a$  has this property as well. We see that the result of `foldl_except` contains all predecessors from  $a$ . By Lemma 8,  $a$  is not reached from a cycle. Using a similar argument, every element reaching  $a$  also satisfies the node condition. ◀

► **Lemma 11** (`dfs_step_extends_verified` and `dfs_step_result_contains_a`). *Consider the input for `verify_via_dfs_step` with  $a$  being the argument for the single vertex. If `verify_via_dfs_step` is successful and returns `verifiedAfter`, then `verifiedNodes`  $\subseteq$  `verifiedAfter`. Furthermore, then  $a \in \text{verifiedAfter}$ .*

**Proof.** We prove this by induction on the number of vertices that occur in  $G$  but not in `currWalk`. The base case is again trivial. For the induction step, we consider the cases of  $a$  already being contained in `verifiedNodes` or not. If yes, the claim follows directly. Otherwise, we note that due to the induction hypothesis, the resulting set will be a superset of the input set. We finally add  $a$  to this set so that the statement is proven. ◀

Using this result, we can complete the missing proofs of Theorem 9 and Theorem 5.

**Proof of Theorem 9.** We show the claim via induction. For the induction step, we show both directions individually. If `verify_via_dfs_step` is successful, then the claim follows from Lemma 11 and Lemma 10. For the other direction, assume that  $a$  has property *DfsStepSemantics*. We show that `verify_via_dfs_step` is successful. If  $a$  is in `verifiedNodes`, the claim follows. Otherwise, we know by property *DfsStepSemantics* that *cond* holds for  $a$ , since  $a$  can reach itself, and that all predecessors of  $a$  are not in `walkFromA`. The claim follows from

the induction hypothesis once we prove that *DfsStepSemantics* holds for every predecessor  $b$  of  $a$  by showing that (1)  $b$  is not reached from a cycle and that (2) for every vertex  $c$  that reaches  $b$ , the node condition holds. Claim (1) holds by assumption and Lemma 8. Claim (2) holds by assumption and since every vertex that can reach  $b$  also can reach  $a$ . ◀

**Proof of Theorem 5.** By definition, `verify_via_dfs` is successful if and only if the calls of `verify_via_dfs_step` are successful for every vertex in  $G$ . Also, by Lemma 7,  $G$  is acyclic if and only if every vertex in  $G$  is not reached by a cycle. Additionally, we note that *cond* holds for every vertex if and only if this holds for any vertex that is reachable from a vertex in the graph because every vertex that is reachable from a vertex is also in  $G$  by the definition of `isWalk` and `canReach`. After applying these two equivalences, the claim follows from Theorem 9. ◀

## 4.4 Ordered Graph Approach

For the `Array`-based ordered graph approach, we reduce the complexity of the previous check by requiring a data structure that is acyclic by definition:

```
abbrev OrderedProofGraph (τ : Signature) := {arr : Array ((GroundAtom τ) × List ℕ)
  // ∀ i : Fin arr.size, ∀ j ∈ arr[i].snd, j < i }
```

To certify that an ordered proof graph is valid for a given knowledge base, we check that each index is *locally valid* i.e. if the vertex label is either in the database when it has no predecessors, or it follows from the labels of its predecessors via a rule.

```
def locallyValid (G : OrderedProofGraph τ) (kb : KnowledgeBase τ)
  (i : Fin G.val.size) : Prop :=
  (G.val[i].snd == [] ∧ kb.db.contains G.val[i].fst)
  ∨ (∃ r ∈ kb.prog, ∃ (g : Grounding τ), g.applyRule' r = {
    head := G.val[i].fst
    body := G.val[i].snd.attach.map (fun n => (G.val.get n.val _).fst))})
```

The validity can be checked locally by `checkAtIndex` similarly to trees and is globally checked by `checkValidity`. Because of the ordering, we can safely assume that all predecessors are already certified when iterating through the indices starting at 0. The `ProofTree` construction can be implemented analogously to the unordered case. Termination follows from the fact that the predecessors of each vertex have strictly smaller indices.

An interesting observation is that the ordered graph check runs in LOGSPACE in the size of the input graph as it only iterates over the array once, whereas the acyclicity for directed graphs that is needed in the unordered case is NLOGSPACE-complete. This illustrates that the actual encoding of certificates can be an important aspect in a certificate-based approach. While computational savings during validation will generally have to be paid for in certificate creation, the more complex computational processes that are to be certified might already include the necessary work naturally. This is likely true in our case: any reasoning engine that produces (valid) proofs should be aware of the ordering that we require, since it is the order of its own derivations. Of course, it is not clear if a theoretical reduction in worst-case complexity is reflected by practical performance without suitable empirical evaluations.

## 5 Model Checking

Besides verifying soundness, we also want to provide basic functionality to check if the atoms included in the input form a (first-order) model of the input program. In this section we

```

def checkPGR (m : CheckableModel  $\tau$ ) (pgr : PartialGroundRule  $\tau$ )
  (safe : pgr.isSafe) : Except String Unit :=
  match eq : pgr.ungroundedBody with
  | .nil => if pgr.head.toGroundAtom _  $\in$  m
    then Except.ok ()
    else Except.error ("Unsatisfied rule: " ++ ToString.toString pgr.toRule)
  | .cons hd tl =>
    (m.substitutionsForAtom hd).attach.mapExceptUnit (fun <s, s_mem> =>
      let adjustedRule : PartialGroundRule  $\tau$  := {
        head := s.applyAtom pgr.head
        groundedBody := pgr.groundedBody ++ [(s.applyAtom hd).toGroundAtom _]
        ungroundedBody := tl.map s.applyAtom}
      m.checkPGR adjustedRule _)

```

■ **Listing 5** Function to check whether an Interpretation is a model

require rules to be safe (`isSafe`), i.e. all variables in the head also occur in the body.

In principle, the implementation is straightforward: collect all atoms from the tree/graph into an interpretation  $i$  and check for all possible rule body instantiations if their head is also present. To improve performance over a brute force approach, we introduce `PartialGroundRules` that allow us to keep track which body atoms we have instantiated while applying the same variable mapping on all other atoms. Any rule can be transformed into a partial ground rule and vice versa so that we may reuse terminology from rules.

```

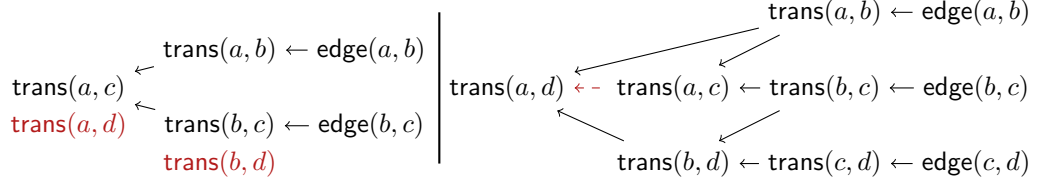
structure PartialGroundRule ( $\tau$ : Signature) where
  head: Atom  $\tau$ 
  groundedBody: List (GroundAtom  $\tau$ )
  ungroundedBody: List (Atom  $\tau$ )

```

Additionally, this structure allows stopping the procedure early. We call a partial ground rule  $r$  active in  $i$ , if all atoms in the grounded body are in  $i$ . If not active, then  $r$  is satisfied for any remaining variable assignment and we can stop early (`satisfied_of_not_active`).

We use this in a recursive exploration using `checkPGR` (see Listing 5). If the ungrounded body is empty, then we simply check if the head, which due to the safety assumption must also be a ground atom, is in  $i$ . If the ungrounded body is not empty, we take the first element and check for all possible substitutions between this element and ground atoms in  $i$ . In order to be consistent we apply this substitution to the whole partial ground rule (except for the grounded body which is not affected by it). On all these new partial ground rules we again call `checkPGR`. In order to check the whole program, we call this procedure for every rule which at the start is always active. This returns the correct result (`checkPGRIsOkIffRuleIsSatisfied`) for any active rule and the newly created rules in the procedure preserve activeness.

Apart from using the activeness of the partial ground rules, the other main idea of the proof is that we can transform groundings that instantiate the first atom of the ungrounded body into a substitution from `substitutionsForAtom` and a grounding and vice versa. Note that this is only a first version for a completeness check for datalog reasoning not yet implementing optimizations that would be necessary for large models.



■ **Figure 2** A proof tree and a proof graph for Example 12. Invalid versions are indicated in red.

## 6 Evaluation

In this section, we evaluate the practicality of our Lean implementation on synthetic and real-world examples. Moreover, we compare the different encodings of datalog proofs in practice. To obtain Datalog consequences and proofs, we use the Datalog reasoning engine Nemo [22], which has a tracing feature that returns proof graphs in a custom JSON format.<sup>2</sup> Transformations of Nemo’s output into our various formats are realized by Python scripts. On top of the implementation described in the previous sections, we created a parsing infrastructure for a JSON-based input format in [Main](#) and [Parsing](#).

Database facts are not part of our JSON representation. Instead, we interpret nodes in proof trees or graphs as database facts if they do not have any predecessors. For production use, an explicit representation of the assumed facts would be preferable, resulting in more robust certificates, but this is not important for our (performance) evaluation. A full validation might also have to encompass that facts and rules are correctly read from the input sources (e.g., CSV files), which is beyond the scope of the certification-based approach.

We illustrate the usage of our tool with the following toy example showing both a valid and an invalid proof tree/graph. The example also reflects the experimental setup.

► **Example 12.** Consider the transitive closure rules (1) and (2) from the introduction together with the facts  $\{\text{edge}(a, b), \text{edge}(b, c), \text{edge}(c, d)\}$ . Using Nemo, we can derive all six entailed facts for  $\text{trans}$ . For each fact, Nemo further provides a *trace* showing how the system has applied the rules to obtain the result. A Python script converts such traces into proof trees or proof graphs in our Nemo-independent JSON format. Figure 2 shows a proof tree for  $\text{trans}(a, c)$  (left), and a proof graph for  $\text{trans}(a, d)$  and  $\text{trans}(a, c)$  (right). We manually add some errors to the proofs, indicated in red in Figure 2.

These valid and invalid proof files are in the directory [tcToyExample](#) of our supplementary material, which also contains further documentation for reproducing this example. Running our prototype checker as documented, we obtain the expected outputs where the original proofs are valid and their modifications are not. Invalid cases also report a short error message, e.g., “invalid result: no match for  $\text{trans}(a, d) :- \text{trans}(a, b), \text{trans}(b, d), \text{trans}(a, c)$ ” for the invalid proof graph in Figure 2 or can also be recognized via the exit code 1. Our other evaluations follow the same pattern and can also be reproduced from the supplementary material.

We consider four scenarios – (1a), (1b), (2), and (3) – which include realistic synthetic and real-world applications of Datalog. Moreover, we consider individual certification of single facts and joint certification of many facts. The latter case is implemented for proof

<sup>2</sup> The Nemo version used was <https://github.com/knowsys/nemo/releases/tag/v0.7.1>



■ **Table 1** Evaluation (times and proof sizes) using trees (T), and unordered/ordered graphs (G/O)

	Nemo	Size(T)	Size(G)	Size(O)	Time(T)	Time(G)	Time(O)
(1a)	59s	320KB	515KB	320KB	0.1s	0.1s	0.1s
(1b)	<0.1s	53MB	1.7MB	564KB	2.7s	0.2s	0.1s
(2)	<0.1s	313MB	16KB	12KB	16s	<0.1s	<0.1s
(3)	7.8s	8.7MB	9.9MB	4.4MB	0.3s	0.4s	0.2s

trees by checking a collection of trees, and for proof graphs by checking a single graph that includes all conclusions that are to be certified.

(1) We consider the rules of Example 12 with facts of the form  $\text{edge}(0, 1), \dots, \text{edge}(n-1, n)$  for some number  $n$ . For (1a), we set  $n = 1000$  and check the conclusion  $\text{trans}(0, 1000)$  (`tcBenchSingleFact`); for (1b), we set  $n = 100$  and check all 5050 conclusions for  $\text{trans}$  (`tcBenchAllFacts`). Transitivity is a frequent type of (sub-)task in Datalog reasoning, but chains of length 1000 are very long even for large datasets, since transitivity is mostly used on hierarchies that are relatively shallow.

(2) We consider the rules of Example 3 with an edge-chain of length  $n = 20$ , and we check the conclusion  $\text{trans}(0, 20)$  (`tcBenchExponential`). This scenario emphasizes the possibly exponential advantage of proof graphs over trees.

(3) We consider the real-world task of reasoning in the OWL EL profile of the W3C Web Ontology Language, which is possible in Datalog by translating reasoning calculi to rules [13]. We use the derivation rules by Kazakov et al. [24], and the medical ontology GALEN that was also used in the evaluation of their work. The rules and pre-processed input ontology<sup>3</sup> yield more than 1.8 million facts. We randomly select 1000 conclusions from the output predicate `mainSubClassOf` that represents the output of the EL reasoning task (`elReasoning`).

All experiments were performed on a mid-range laptop (Intel Core i5 gen8, 16GB RAM, NixOS Linux). Table 1 reports the results including Nemo reasoning times without tracing. The other columns give the proof file sizes and total validation time for the scenarios representing proofs as trees (T), unordered graphs (G), or ordered graphs (O). Times are overall wall-clock times of the prototype checker averaged over 5 runs, and sizes refer to compact (not pretty-printed) JSON. For example, while Nemo reasons in (1b) for less than 0.1 seconds, the proof trees amount to 53MB of JSON taking 2.7 seconds to validate.

Overall, even the check based on unordered graphs is very fast in all cases and the tree-based implementation achieves similar performance in cases (1a) and (3). The overall fast times for (3) reflect the fact that proofs in real-world applications are rarely very deep, even in cases where millions of consequences are derived. The advantages of graphs are seen when considering many overlapping proofs (1b) and, as expected, for rules that realize the exponential worst-case penalty of using proof trees (2). The ordered graph approach outperforms the others in all scenarios regarding both time and memory.

We still find that both tree-based and graph-based validations are fit for practical use. While graphs are theoretically and empirically superior, the choice may also be governed by the native tracing output of a tool (which is graph-like in Nemo, but tree-like, e.g., in Soufflé). While the ordered graph would always be the best option, it also requires some

<sup>3</sup> <https://github.com/knownsys/nemo-examples/tree/main/examples/owl-el/from-preprocessed-csv>

additional effort to output the trace in the right order. But when all inferences have indeed been done properly by the reasoning engine, this should not be too much of a challenge.

## 7 Related Work

The *de Bruijn* criterion [6] requires that the software tools (e.g., logical reasoners) provide certificates together with their output and the tool checking the validity based on output and certificate (e.g., our Lean-based validator) are independent entities. Besides certificate-based verifiers [5, 21], a similar criterion has been coined as *certifying algorithms* in software engineering [29]. One approach there is to transpile certificate checker code written in the C programming language to an interactive theorem prover. Alkassar et al. [2] use a transpilation to Isabelle to verify the certificate checker in this sense. Thereby, one has to additionally trust the transpilation while our work implements the checker directly in Lean. Baader et al. [4] generate certificates for consequences of the *description logic*  $\mathcal{EL}$ , computed by an existing  $\mathcal{EL}$ -reasoner. The validation of the generated proof certificates is done by the LSFC checker<sup>4</sup>. In contrast to this approach, we have provided the correctness of our proof validator (w.r.t. the Datalog semantics) ourselves and, therefore, do not rely on third-party developments like the LSFC checker. Our approach partly generalizes the work of Baader et al. in that the  $\mathcal{EL}$ -reasoning calculus can be expressed in Datalog [25], making our proof validator also available to  $\mathcal{EL}$  reasoning.

Another approach to building trust in automated reasoning engines is to implement the reasoner itself (i.e., not a certificate checker) in Lean. For instance, Benzaken et al. [7] and Dumbravă [16] proposed a certified Datalog engine written in Rocq. Later on, a similar approach was implemented for a subset of Datalog, which Bonifati et al. [8] call *Regular Datalog*, for (knowledge) graph view maintenance. The Rocq implementations formalize the model-theoretic semantics up to basic definitions for rule satisfaction and modelhood, but fully integrate the fixed-point semantics for proving correctness. Whitehead [36] also formalizes the fixed point semantics of Datalog and its extension *Binder*, a security logic, in Rocq to extract a monitoring framework for access control policies. An Isabelle/HOL formalization of the model-theoretic semantics of *stratified Datalog* – an non-monotone extension of Datalog – is provided by Schlichtkrull et al. [33] to verify program analysis algorithms, which are themselves written in Datalog. In contrast, our work fully formalizes the model-theoretic semantics as well as the proof-theoretic semantics, neglecting the fixed point semantics. Thereby, we aim at independence of our implementation from the reasoners in use (i.e., ultimately satisfying the *de Bruijn* criterion). This independence additionally preserves the original reasoners’ efficiency.

A rather data-driven approach has been coined as *data provenance* [10, 20], providing *explanations* in different formats (formalized as expressions of certain semirings). Such explanations are often based on the (finite) input database. Notably, usual provenance explanations for Datalog reasoning results are obtained from analyzing all (i.e., infinitely many) proof trees [17] or a careful selection thereof [11]. Thus, our work provides the necessary ingredients to obtain a fully formalized theory of data provenance for Datalog. While provenance solutions, specialized for particular reasoners [32], exist, our certificate-based framework is independent of the reasoner in use, as long as it provides proof trees (i.e., certificates) for their derivations.

---

<sup>4</sup> <https://github.com/CVC4/LFSC>

## 8 Conclusion and Future Work

We propose a certificate-based approach to the verification of Datalog reasoning, built on the proof-theoretic semantics of Datalog. The certificate checker is formalized and implemented in Lean, validating proof trees of Datalog conclusions, as provided by Datalog tools such as Nemo [22] or Soufflé [23]. Our evaluation shows that the exponentially more succinct proof graphs can accelerate the validation of complex derivations or multiple conclusions.

In practical applications, Datalog is often extended with further features, which should be covered in future works. Relevant extensions include: (1) negation in rule bodies, stratified or under a general semantics for normal logic programs; (2) existential quantification in rule heads [12]; (3) datatypes and aggregate functions; and (4) function terms and complex values, such as tuples or sets [1, 28]. Function terms, complex values, and existential quantifiers are closest to our work, since they admit similar proof structures. With negation and aggregates, proofs must refer to all facts of a certain shape, e.g., to show that a conclusion was *not* derived, which can lead to larger proofs and more checks. Stratified negation has already been integrated in certified reasoners [7, 8], but certificate checkers for stratified Datalog do not yet exist. Datatypes in turn present their own challenges, but their validation could be based on existing support for some datatypes in Lean.

Within a Datalog toolchain, our tool builds trust in these database systems with the ability to exemplarily verify derivations. In principle, one could automatically check the proof trees for each derivation. In many applications this might not be viable if there is a large number of facts and is likely also not necessary once we can be sure enough that the system indeed works as intended after enough exemplary checks. However, for critical applications, one could directly generate certificates, i.e. proof trees/graphs, for each derived fact and automatically check their correctness with our tool. When debugging a Datalog program or even the system itself, our certified checker can act as a companion to explainability tools by checking that the given explanation in form of a proof tree/graph is valid in the first place, which enhances user trust into the explanation and the result of the Datalog system. Another line of research is to directly verify the Datalog reasoner itself but, as mentioned in the introduction, this is challenging since today's Datalog systems are highly sophisticated and optimized pieces of software. Reimplementing the systems in e.g. Lean most likely cannot compete with implementations directly done in C(++) or Rust in terms of performance. However, it might be possible (with significant effort) to maintain both implementations (the formally verified one and the fast one) and verify with enough tests that their components behave the same on all inputs. Automatic transpilations could also come in handy here.

---

## References

- 1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1994.
- 2 Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, and Christine Rizkallah. Verification of certifying computations. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd Int. Conf. CAV'11*, volume 6806 of *LNCS*, pages 67–82. Springer, 2011. doi:10.1007/978-3-642-22110-1\_7.
- 3 Mauricio Ayala-Rincón and César A. Muñoz, editors. *Proc. 8th Int. Conf. on Interactive Theorem Proving (ITP'17)*, volume 10499 of *LNCS*. Springer, 2017.
- 4 Franz Baader, Patrick Koopmann, and Cesare Tinelli. First results on how to certify subsumptions computed by the EL reasoner ELK using the logical framework with side conditions. In Stefan Borgwardt and Thomas Meyer, editors, *Proc. 33rd Int. Workshop on Description*

- Logics (DL 2020)*, volume 2663 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2020. URL: <https://ceur-ws.org/Vol-2663/paper-5.pdf>.
- 5 Seulkee Baek. A Formally Verified Checker for First-Order Proofs. In Liron Cohen and Cezary Kaliszyk, editors, *Proc. 12th Int. Conf. on Interactive Theorem Proving (ITP'21)*, volume 193 of *LIPIcs*, pages 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.6.
  - 6 Henk Barendregt and Freek Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1835):2351–2375, 2005. doi:10.1098/rsta.2005.1650.
  - 7 Véronique Benzaken, Evelyne Contejean, and Ștefania-Gabriela Dumbravă. Certifying standard and stratified datalog inference engines in ssreflect. In Ayala-Rincón and Muñoz [3], pages 171–188. doi:10.1007/978-3-319-66107-0\_12.
  - 8 Angela Bonifati, Ștefania-Gabriela Dumbravă, and Emilio Jesús Gallego Arias. Certified graph view maintenance with regular datalog. *Theory Pract. Log. Program.*, 18(3-4):372–389, 2018. doi:10.1017/S1471068418000224.
  - 9 Florent Bréhard, Assia Mahboubi, and Damien Pous. A Certificate-Based Approach to Formally Verified Approximations. In John Harrison, John O’Leary, and Andrew Tolmach, editors, *Proc. 10th Int. Conf. on Interactive Theorem Proving (ITP’19)*, volume 141 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.8.
  - 10 Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Proc. 8th Int. Conf. on Database Theory (ICDT’01)*, volume 1973 of *LNCS*, pages 316–330. Springer, 2001.
  - 11 Marco Calautti, Ester Livshits, Andreas Pieris, and Markus Schneider. The complexity of why-provenance for datalog queries. *Proc. ACM Manag. Data*, 2(2):83, 2024. doi:10.1145/3651146.
  - 12 Andrea Cali, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In Jan Paredaens and Jianwen Su, editors, *Proc. 28th Symposium on Principles of Database Systems (PODS’09)*, pages 77–86. ACM, 2009.
  - 13 David Carral, Markus Krötzsch, and Jacopo Urbani. Practical uses of existential rules in knowledge representation. Tutorial at the 24th European Conference on Artificial Intelligence (ECAI’20); lecture materials online at [https://iccl.inf.tu-dresden.de/web/Rules\\_ECAI\\_Tutorial\\_2020/en](https://iccl.inf.tu-dresden.de/web/Rules_ECAI_Tutorial_2020/en), 2020.
  - 14 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer, 1990.
  - 15 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
  - 16 Ștefania-Gabriela Dumbravă. *Formalisation en Coq de Bases de Données Relationnelles et Déductives -et Mécanisation de Datalog*. PhD thesis, Université Paris Saclay (COMUE), 2016.
  - 17 Ali Elhalawati, Markus Krötzsch, and Stephan Mennicke. An existential rule framework for computing why-provenance on-demand for datalog. In Guido Governatori and Anni-Yasmin Turhan, editors, *Proc. 2nd Int. Joint Conf. on Rules and Reasoning (RuleML+RR’22)*, volume 13752 of *LNCS*, pages 146–163. Springer, 2022.
  - 18 Asta Halkjær From and Frederik Krogsdal Jacobsen. Verifying a Sequent Calculus Prover for First-Order Logic with Functions in Isabelle/HOL. In June Andronick and Leonardo de Moura, editors, *Proc. 13th Int. Conf. on Interactive Theorem Proving (ITP’22)*, volume 237 of *LIPIcs*, pages 13:1–13:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.13.
  - 19 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019. doi:10.1017/S1471068418000054.
  - 20 Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40, 2007.

- 21 Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Ayala-Rincón and Muñoz [3], pages 269–284. doi:10.1007/978-3-319-66107-0\_18.
- 22 Alex Ivliev, Lukas Gerlach, Simon Meusel, Jakob Steinberg, and Markus Krötzsch. Nemo: Your Friendly and Versatile Rule Reasoning Toolkit. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, pages 743–754, 8 2024. doi:10.24963/kr.2024/70.
- 23 Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th Int. Conf. (CAV'16)*, volume 9780 of *LNCS*, pages 422–430. Springer, 2016. doi:10.1007/978-3-319-41540-6\_23.
- 24 Yevgeny Kazakov, Markus Krötzsch, and František Simančík. The incredible ELK: From polynomial procedures to efficient reasoning with  $\mathcal{EL}$  ontologies. *J. of Automated Reasoning*, 53:1–61, 2013.
- 25 Markus Krötzsch. Efficient rule-based inferencing for OWL EL. In Toby Walsh, editor, *Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI'11)*, pages 2668–2673. AAAI Press/IJCAI, 2011.
- 26 Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholtz. Metamorphic testing of datalog engines. In *Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*, page 639–650. ACM, 2021. doi:10.1145/3468264.3468573.
- 27 Muhammad Numair Mansur, Valentin Wüstholtz, and Maria Christakis. Dependency-aware metamorphic testing of datalog engines. In *Proc. 32nd ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA'23)*. ACM, 2023. doi:10.1145/3597926.3598052.
- 28 Maximilian Marx and Markus Krötzsch. Tuple-generating dependencies capture complex values. In Dan Olteanu and Nils Vortmeier, editors, *Proc. 25th Int. Conf. on Database Theory (ICDT'22)*, volume 220 of *LIPICs*, pages 13:1–13:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ICDT.2022.13.
- 29 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011. doi:10.1016/J.COSREV.2010.09.009.
- 30 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635. Springer, 2021.
- 31 Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A highly-scalable RDF store. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul T. Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II*, volume 9367 of *LNCS*, pages 3–20. Springer, 2015.
- 32 Yann Ramusat, Silviu Maniu, and Pierre Senellart. Efficient provenance-aware querying of graph databases with datalog. In *Proc. 5th ACM SIGMOD Joint Int. Workshop on Graph Data Management Experiences & Systems and Network Data Analytics (GRADES-NDA'22)*. ACM, 2022. doi:10.1145/3534540.3534689.
- 33 Anders Schlichtkrull, Rene Rydhof Hansen, and Flemming Nielson. Isabelle-verified correctness of datalog programs for program analysis. In *Proc. 39th ACM/SIGAPP Symposium on Applied Computing (SAC 2024)*, pages 1731–1734. ACM, 2024. doi:10.1145/3605098.3636091.
- 34 Jacopo Urbani, Cerial Jacobs, and Markus Krötzsch. Column-oriented Datalog materialization for large knowledge graphs. In Dale Schuurmans and Michael P. Wellman, editors, *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16)*, pages 258–264. AAAI Press, 2016.
- 35 Niels van der Weide, Deivid Vale, and Cynthia Kop. Certifying Higher-Order Polynomial Interpretations. In Adam Naumowicz and René Thiemann, editors, *Proc. 14th Int. Conf. on Interactive Theorem Proving (ITP'23)*, volume 268 of *LIPICs*, pages 30:1–30:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ITP.2023.30.

- 36   Nathan Whitehead. A certified distributed security logic for authorizing code. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, pages 253–268. Springer, 2007. doi:10.1007/978-3-540-74464-1\_17.
- 37   Chi Zhang, Linzhang Wang, and Manuel Rigger. Finding Cross-Rule Optimization Bugs in Datalog Engines. *Proc. ACM Program. Lang.*, 8(OOPSLA1):110–136, 2024. doi:10.1145/3649815.