Verifying Datalog Reasoning with Lean

Johannes Tantow[1] , **Lukas Gerlach**[2] , **Stephan Mennicke**[2] , **Markus Krötzsch**[2]
[1]TU Chemnitz, Germany
[2]Knowledge-Based Systems Group, TU Dresden, Germany
johannes.tantow@informatik.tu-chemnitz.de,
{lukas.gerlach, stephan.mennicke, markus.kroetzsch}@tu-dresden.de

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, 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 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.

1 Introduction

Datalog is a simple yet elegant rule language that is at the heart of many approaches to logic programming, knowledge representation, and data analysis (Abiteboul, Hull, and Vianu 1994; Ceri, Gottlob, and Tanca 1990). 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 quantifiers, as is customary in logic programming):

$$\mathsf{trans}(x, y) \leftarrow \mathsf{edge}(x, y) \tag{1}$$
$$\mathsf{trans}(x, z) \leftarrow \mathsf{trans}(x, y) \wedge \mathsf{trans}(y, z) \tag{2}$$

The main reasoning task is to compute all logical consequences of such *programs* over a given set of facts F. Deciding such consequences is P-complete in the size of F, making Datalog interesting for data-intensive applications.

Datalog is well-supported in practice. Modern rule engines such as *clingo* (Gebser et al. 2019), *Nemo* (Ivliev et al. 2024), *RDFox* (Nenov et al. 2015), *Soufflé* (Jordan, Scholz, and Subotic 2016), and *VLog* (Urbani, Jacobs, and Krötzsch 2016) easily compute millions of 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 language extensions and rely on intricate optimizations.

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 (Mansur, Christakis, and Wüstholz 2021; Mansur, Wüstholz, and Christakis 2023; Zhang, Wang, and Rigger 2024). Stronger correctness guarantees are given by certified Datalog systems, but the prototypes that exist cannot scale to the input sizes of optimized systems yet (Benzaken, Contejean, and Dumbravă 2017), or restrict to a subset of Datalog (Bonifati, Dumbravă, and Arias 2018).

In this paper, we address this challenge by developing a certificate checker for Datalog, written and verified in Lean 4 (de Moura and Ullrich 2021), a purely functional programming language that acts as an interactive theorem prover but also allows to compile executable code. 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 (Barendregt and Wiedijk 2005)) exist in various fields (Heule et al. 2017; Baek 2021; From and Jacobsen 2022; van der Weide, Vale, and Kop 2023; Bréhard, Mahboubi, and Pous 2019), 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 *proof trees* for their derivations.

The task for our checker is to take such a proof tree as input and, given the underlying Datalog program, to check if the proof tree is indeed correct. We allow our checker to ingest JSON-formatted inputs that encode Datalog rules and proofs in an implementation-independent exchange format. Besides the traditional *proof trees* encoding, we consider a redundancy-avoiding encoding as *proof graphs* that can be exponentially more succinct. We evaluate the feasibility and performance using synthetic and real-world Datalog tasks and proofs generated by the Datalog engine Nemo.

In this extended abstract of our full paper (Tantow et al. 2025), we give an overview of the formalization and implementation of our verified Datalog proof checker. Many Lean symbols link to their formal definitions in our repository.[1]

[1]https://github.com/knowsys/CertifyingDatalog/tree/v0.2.0

	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

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

2 Verifying Different Proof Formats

We start by formalizing the basic building blocks of the syntax of Datalog based from terms to programs in `Basic.lean`. As the mathematical baseline for the formal verification of the implementation of our checker, we define the `modelTheoreticSemantics` as the set of all ground atoms that occur in every model and prove this set to be the least model. Additionally, we define the `proofTheoreticSemantics` as the set of all atoms that have a valid `ProofTree`. A `ProofTree` is a standard inductive tree definition labeled with ground atoms (a `ProofTreeSkeleton`) together with a predicate that describes its validity. We can inductively define when such a `ProofTreeSkeleton.isValid` for a knowledge base by asserting the existence of an appropriate rule in each step. To increase trust in this formal baseline, we prove that both semantics are equivalent, as they should be, in `modelAndProofTreeSemanticsEquivalent`. Our checker implementation will consequently be verified against the `proofTheoreticSemantics`.

The straightforward check, `checkValidity`, takes in a `ProofTreeSkeleton` and outputs whether it is valid or not. This check is implemented in such a way that Lean can compile it into executable code, which would not be possible for the mathematical description given in `ProofTreeSkeleton.isValid`. We then only need to prove that `checkValidity` indicates success if and only if the tree is indeed valid with respect to the formal definition: `checkValidityOkIffIsValid`. One of the main challenges here is to find rules that match the steps in the tree, which requires a basic unification algorithm.

In Datalog proofs where atoms are used multiple times in the derivation, it can be beneficial to represent derivations not as trees but as directed acyclic graphs. With this approach, we also aim to boost performance of our validation procedure. For this, we use adjacency lists to associate each node with a list of its predecessors. We propose two encodings: the first, called *unordered*, encodes the adjacency list as a hash map (see `Graph`); the second, called *ordered*, uses an array instead (see `OrderedProofGraph`). The second approach gets its name because it requires the input to be topologically sorted, i.e. added nodes may only reference nodes as predecessors that have already been added to the array before. In particular, the indices of the predecessors of a node are always smaller than the node's index. For the ordered approach, this directly guarantees acyclicity of the encoded graph. For the unordered approach, this is not necessarily the case and needs to be validated.

Both validations have the following essential ingredients. We put hyperlinks to the respective implementations by "O" (ordered) and "U" (unordered). (1) they allow the extraction of `ProofTreeSkeleton`s (`O`/`U`), (2) they have a mathematical description for what it means for them to be valid (`O`/`U`), (3) there is a proof showing that their validity according to (2) implies validity of their skeleton from (1) (`O`/`U`), (4) they have a check for validity that compiles to executable code (`O`/`U`), and (5) the check from (4) indicates success if and only if the graph is valid according to (2) (`O`/`U`).

Because of the nature of the ordered approach, all these parts are relatively straightforward. The unordered approach requires machinery to implement and verify a depth-first search through the graph that checks (4) but also ensures graph acyclicity. We refer the interested reader to our full paper (Tantow et al. 2025) for details.

3 Evaluation

To obtain Datalog consequences and proofs for our evaluation, we use the Datalog reasoning engine Nemo (Ivliev et al. 2024), which has a tracing feature that returns proof graphs in a custom JSON format.[2] Nonetheless, our checker is engine-agnostic and can work with any engine that returns traces as some sort of graph or tree. The result of the engine has to be submitted in a simple JSON-encoding as seen in `examples`. We consider three scenarios (1a), (1b), (2) that are based on computing the transitive closure of a relation where (2) is extended to create exponential overhead for representing proofs as trees as opposed to graphs. Another example (3) uses the real-world task of reasoning with OWL EL ontologies, which is possible in pure Datalog. See our paper (Tantow et al. 2025) or repository for details.

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 JSON.

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 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.

[2]The Nemo version used was https://github.com/knowsys/nemo/releases/tag/v0.7.1

Acknowledgments

This work was partly supported by Deutsche Forschungsgemeinschaft (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).

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

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1994. *Foundations of Databases*. Addison Wesley.

Ayala-Rincón, M., and Muñoz, C. A., eds. 2017. *Proc. 8th Int. Conf. on Interactive Theorem Proving (ITP'17)*, volume 10499 of *LNCS*. Springer.

Baek, S. 2021. A Formally Verified Checker for First-Order Proofs. In Cohen, L., and Kaliszyk, C., eds., *Proc. 12th Int. Conf. on Interactive Theorem Proving (ITP'21)*, volume 193 of *LIPIcs*, 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Barendregt, H., and Wiedijk, F. 2005. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 363(1835):2351–2375.

Benzaken, V.; Contejean, E.; and Dumbravă, Ş.-G. 2017. Certifying standard and stratified datalog inference engines in ssreflect. In Ayala-Rincón and Muñoz (2017), 171–188.

Bonifati, A.; Dumbravă, Ş.-G.; and Arias, E. J. G. 2018. Certified graph view maintenance with regular datalog. *Theory Pract. Log. Program.* 18(3-4):372–389.

Bréhard, F.; Mahboubi, A.; and Pous, D. 2019. A Certificate-Based Approach to Formally Verified Approximations. In Harrison, J.; O'Leary, J.; and Tolmach, A., eds., *Proc. 10th Int. Conf. on Interactive Theorem Proving (ITP'19)*, volume 141 of *LIPIcs*, 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Ceri, S.; Gottlob, G.; and Tanca, L. 1990. *Logic Programming and Databases*. Springer.

de Moura, L., and Ullrich, S. 2021. The lean 4 theorem prover and programming language. In Platzer, A., and Sutcliffe, G., eds., *Automated Deduction – CADE 28*, 625–635. Springer.

From, A. H., and Jacobsen, F. K. 2022. Verifying a Sequent Calculus Prover for First-Order Logic with Functions in Isabelle/HOL. In Andronick, J., and de Moura, L., eds., *Proc. 13th Int. Conf. on Interactive Theorem Proving (ITP'22)*, volume 237 of *LIPIcs*, 13:1–13:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19(1):27–82.

Heule, M.; Hunt, W.; Kaufmann, M.; and Wetzler, N. 2017. Efficient, verified checking of propositional proofs. In Ayala-Rincón and Muñoz (2017), 269–284.

Ivliev, A.; Gerlach, L.; Meusel, S.; Steinberg, J.; and Krötzsch, M. 2024. Nemo: Your Friendly and Versatile Rule Reasoning Toolkit. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, 743–754.

Jordan, H.; Scholz, B.; and Subotic, P. 2016. Soufflé: On synthesis of program analyzers. In Chaudhuri, S., and Farzan, A., eds., *Computer Aided Verification - 28th Int. Conf. (CAV'16)*, volume 9780 of *LNCS*, 422–430. Springer.

Mansur, M. N.; Christakis, M.; and Wüstholz, V. 2021. Metamorphic testing of datalog engines. In *Proc. of the 29th ACM Joint Meeting on European Software EngineeringConf. and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*, 639–650. ACM.

Mansur, M. N.; Wüstholz, V.; and Christakis, M. 2023. Dependency-aware metamorphic testing of datalog engines. In *Proc. 32nd ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA'23)*. ACM.

Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; and Banerjee, J. 2015. RDFox: A highly-scalable RDF store. In Arenas, M.; Corcho, Ó.; Simperl, E.; Strohmaier, M.; d'Aquin, M.; Srinivas, K.; Groth, P. T.; Dumontier, M.; Heflin, J.; Thirunarayan, K.; and Staab, S., eds., *Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II*, volume 9367 of *LNCS*, 3–20. Springer.

Tantow, J.; Gerlach, L.; Mennicke, S.; and Krötzsch, M. 2025. Verifying datalog reasoning with lean. In Forster, Y., and Keller, C., eds., *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Urbani, J.; Jacobs, C.; and Krötzsch, M. 2016. Column-oriented Datalog materialization for large knowledge graphs. In Schuurmans, D., and Wellman, M. P., eds., *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16)*, 258–264. AAAI Press.

van der Weide, N.; Vale, D.; and Kop, C. 2023. Certifying Higher-Order Polynomial Interpretations. In Naumowicz, A., and Thiemann, R., eds., *Proc. 14th Int. Conf. on Interactive Theorem Proving (ITP'23)*, volume 268 of *LIPIcs*, 30:1–30:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Zhang, C.; Wang, L.; and Rigger, M. 2024. Finding Cross-Rule Optimization Bugs in Datalog Engines. *Proc. ACM Program. Lang.* 8(OOPSLA1):110–136.