Where Does Proof Search Spend Its Effort? A Visualization and Profiling Tool for Formal NLI

Koharu Saeki
Ochanomizu University
saeki.koharu@is.ocha.ac.jp

Daisuke Bekki
Ochanomizu University
bekki@is.ocha.ac.jp

Abstract

Recent work has substantially accelerated proof search in interactive theorem provers by integrating learned models, for both Lean and Coq. The natural language inference (NLI) counterpart lacks an analogous infrastructure: the behavior of dedicated DTT-based provers such as *wani*, inside the Japanese NLI system *lightblue*, is observable today only through verbose textual logs. This opacity blocks ML-acceleration efforts such as *Neural Wani* that need to know where the search spends its time and why it fails. We present a profiling and visualization tool for *wani*, implemented as a web-based component of the *lightblue* development environment, that exposes the proof search through a four-panel dashboard, Search Tree, Flame Graph, Rule Statistics, and Failure Analysis, each making one aspect of prover behavior directly inspectable. The tool provides the observability that ML-acceleration research in NLI currently needs but cannot easily obtain. It is released as open-source software and provided as a Docker image.

1 Introduction

Recent progress in automating proof search in ITPs like Coq (The Coq Development Team, 2024) and Lean (Moura and Ullrich, 2021) leverages machine learning. Approaches like LeanDojo (Yang et al., 2023) and Graph2Tac (Blaauwbroek et al., 2024) integrate learned models into tactic prediction, significantly reducing formalization costs. Extending these results to natural language inference (NLI) is a key challenge. Unlike theorem-proving in mathematics, NLI must cope with linguistic phenomena such as anaphora resolution and presupposition binding, which demand a semantically rich formal framework. Dependent Type Semantics (DTS, Bekki and Mineshima, 2017) provides such a framework, grounded in Dependent Type Theory (DTT, Martin-Löf, 1984). In DTS, establishing an entailment amounts to constructing, from the premises, a proof term that inhabits the type of the hypothesis; the entailment therefore comes with the proof term itself as an explicit witness, not merely a label or score. A representative implementation is *lightblue* (Tomita et al., 2025), with its built-in prover *wani* (Daido and Bekki, 2020). However, NLI proof search involves rapid subgoal proliferation and complex branching, making the search process opaque in practice. Developers currently rely on manual inspection of voluminous text logs, hindering both system improvement and neuro-symbolic efforts like *Neural Wani* (Miyagawa et al., 2026), which aims to accelerate rule selection via ML.

We introduce a profiling tool for DTT-based NLI provers, implemented as a web-based interface for inspecting *lightblue*'s proof-search results. The tool surfaces prover behavior through four coordinated views: *Search Tree*, *Flame Graph*, *Rule Statistics*, and *Failure Analysis*. By making internal processes inspectable, the tool supports both human debugging and ML-driven acceleration. The instrumentation specific to *wani* is a single optional event log; the surrounding design (the structured event schema and the post-hoc reconstruction of the four views) is prover-agnostic and could be applied to other proof-search-based NLI systems, such as the Coq-based ccg2lambda pipeline (Mineshima et al., 2015; Martínez-Gómez et al., 2016), by instrumenting their provers. We release the tool as open-source software and provide it as a Docker image.[1]

Our contributions are as follows:

- We present the first tool for profiling and visualizing proof search in DTT-based NLI provers.
- We demonstrate, on JSeM (Kawazoe et al., 2015), a curated Japanese NLI benchmark,

[1] https://wani-viz-site.fly.dev/

Proceedings of the 6th Workshop on Natural Language Meets Logic and Machine Learning (NALOMA), pages 50–59

that the tool reveals concrete bottlenecks in rule exploration.

- We provide a structured event logging framework that serves as a training-data generator for ML-based acceleration methods such as *Neural Wani*, and release it as open-source software.

2 Background

2.1 Proof Search in Formal NLI

In the DTS framework, the meaning of a sentence is represented as a type. Based on the Curry–Howard correspondence, *wani* decides entailment by searching for a proof of the hypothesis from the premises. Given a context Γ constructed from the premises $S_1 : M_1, \ldots, S_{n-1} : M_{n-1}$, it attempts to construct a term H such that $\Gamma \vdash H : M_n$, where M_n is the type corresponding to the hypothesis. If such a term is found, the entailment holds. Similarly, contradictions are detected by searching for a proof of the negation of the hypothesis, i.e., a term of type $\neg M_n$ (implemented as $M_n \to \bot$). *wani* performs proof search using both forward and backward chaining. Forward chaining derives consequences from the current context, while backward chaining focuses on the goal and recursively decomposes it into subgoals required for rule application. In terms of implementation, given a goal type, the recursive function `deduce'` applies inference rules and attempts to solve the generated subgoals. Specifically, `deduce'` tries each of fourteen inference rules (ΠI, ΣI, ΠE, MEMBERSHIP, etc.) and recurses on the subgoals each accepted rule produces. A few rules (notably MEMBERSHIP and ΠE) invoke forward reasoning internally to discharge subgoals directly from the context or signature. Search is implemented as iterative-deepening depth-first search, where depth-bounded DFS is repeatedly executed with increasing depth limits, under three constraints: a depth bound, a time bound, and loop avoidance via a list of previously failed goals. A rule application can produce several alternative SubGoalSets, each consisting of an indexed list of subgoals that must all be discharged. Because a subgoal may admit multiple solutions, each subgoal may be explored in several alternative branches. As a running example, take problem 720 from JSeM: the entailment "*Taro was cried on by Hanako*" $\to$ "*Hanako cried*". The *lightblue* pipeline yields the proof search query in Figure 1. *wani* succeeds, finding two terms that inhabit the goal type. The type dependencies in G_h determine that its entity, cry, and tense components must be constructed from the corresponding components of the premise variable s_0. Consequently, the meaning representation of "Hanako cried" is already contained as part of the meaning representation of the premise "Taro was cried on by Hanako". Appendix B details the logical constraints that govern this construction.

2.2 The Need for Profiling

Figure 2 shows a typical excerpt of *wani*'s textual debug log, the sole existing means by which a developer can inspect the prover's behavior. From it, individual goals and rule applications are readable. What is not readable by eye is the aggregate, cross-cutting picture: the overall structure of the search, where wall-clock time is spent, which rules dominate the cost, and the patterns by which the prover repeatedly fails. Recovering this picture amounts to providing the observability whose absence Section 1 identified as the practical barrier between current dedicated NLI provers and ML-driven acceleration.

3 The Profiling Tool

Our tool exposes *wani*'s proof search through four coordinated views, Search Tree, Flame Graph, Rule Statistics, and Failure Analysis, each designed to answer a different question about prover behavior. All four views are reconstructed post-hoc from a structured event log embedded inside *wani*.

The tool is deployed as a live demo[1] where the Docker image and installation instructions are also available. The demo presents the four interactive views for JSeM 720, the running example of this paper. A usage guide is provided on the demo site.

3.1 Log Structuring

wani's textual debug output is not produced by a unified logging system. Instead, it consists of ad-hoc `Debug.Trace` `print` calls that developers inserted to inspect specific code paths. The profiler runs alongside these and provides a structured alternative: it adds one field to *wani*'s settings record, an optional reference to a log buffer, and, when present, records timestamped, structured events into an in-memory sequence at each step of `deduce'`. When absent, every logging call short-circuits to a no-op, leaving regular runs unaffected.

The fourteen event kinds fall into four groups (Table 1): goal lifecycle, rule trial, terminal out-

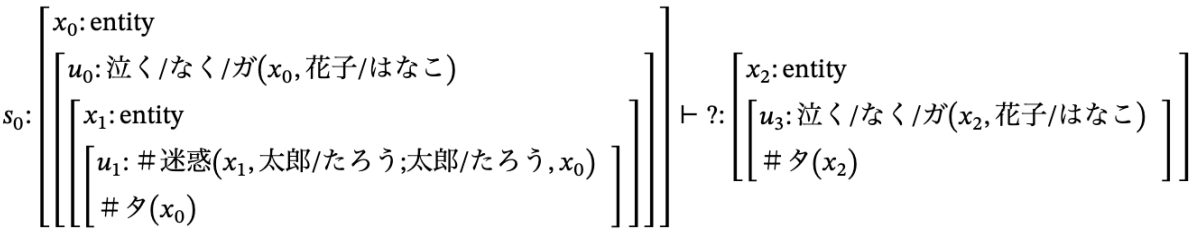

Figure 1: Proof search query for JSeM 720. The premise s_0 corresponds to *Taro was cried on by Hanako*; the goal, to *Hanako cried*.

```
0 current goal : G |- ? : (S entity)(S cry(hanako,0))(S tense(1))T
0 (S entity)(S cry(hanako,0))(S tense(1))T is not acceptable type in PiF
0 (S entity)(S cry(hanako,0))(S tense(1))T is not acceptable type in SigmaF
0 (S entity)(S cry(hanako,0))(S tense(1))T is not acceptable type in IqF
0-acceptable [SubGoalSet SigmaI ...]
0-acceptable [SubGoalSet EFQ ...]
  with SigmaI, want to prove [SubGoal entity, SubGoal S cry. S tense]
  1 current goal : G |- ? : entity
  1 depth @ deduce : entity
0 deduce failed
```

Figure 2: Excerpt of *wani*'s textual debug log on JSeM 720 (depth-1 iteration, failed). *S* denotes Σ; predicate and entity names transliterated from Japanese for readability.

Group	Event kinds
Goal lifecycle	GOALSTART, GOALEND
Rule trial	RULEATTEMPT, RULEACCEPT, RULEREJECT, RULEEXIT
Outcome	DEDUCED, DEDUCEFAILED, DEPTHEXCEEDED, AVOIDLOOP, TIMELIMIT, SPECIALCASE
Search control	ITERDEEPEN, GOALUPDATE

Table 1: The fourteen event kinds emitted by the instrumented prover, grouped by role.

come, and search control. Each event carries content fields (recursion depth, timestamp, goal string, optional rule name, and a message) together with bookkeeping fields used for post-hoc tree reconstruction (event id, kind, goal id, and optional subgoal index within the parent's SubGoalSet).

Aggregation is entirely post-hoc: pairing GOALSTART and GOALEND by goal id, inferring parent–child structure from recursion depth, and grouping events by rule name are performed outside *wani* by the SearchLog module. This separation lets the prover emit events linearly while each view derives the shape it needs (tree, flat list, histogram) on demand.

3.2 Proof Search Tree

The Search Tree visualizes the recursion structure of *wani*'s `deduce'` calls directly. Each node corresponds to one `deduce'` invocation (one goal), and parent-to-child edges encode the subgoals produced by an applied rule. Sibling nodes exhibit three distinct relations. (i) Nodes with different subgoal indices within the same SubGoalSet form an AND relation, where all must succeed. (ii) Nodes corresponding to alternative branches for the same subgoal index form an OR relation, where any one branch suffices. (iii) Nodes from different SubGoalSets, produced by different rules or context bindings, also form an OR relation. The node label `Rule (i/m#k) [children] duration` encodes multiple pieces of information in a compact form. It specifies the applied rule (ΣI, VAR, etc.), the AND position `i/m` (the i-th subgoal out of m produced by the parent rule), and an optional OR index `#k`, which appears when multiple branches exist. It also shows the total number of direct children of this node (including any that are currently collapsed) and the subtree's wall-clock duration in ms. For instance, ΣI (2/4 #1) [10] 0.7 ms denotes the 2nd of four ΣI subgoals, branch 1 of multiple alternatives, with 10 direct children and 0.7 ms duration. Nodes are colored by outcome (green for success, red for failure, orange for depth-bound exhaustion, yellow for loop avoidance, gray for time limit). For scalability on large traces, nodes at depth four or deeper are collapsed by default and can be expanded interactively.

Figure 3 shows the Search Tree for our running example, JSeM 720. The root corresponds to the

Rule	Att.	Suc.	Fail	Dep.	Loop	Time	Pend.	Rate	Tot. ms
CON	12	12	0	0	0	0	0	100%	0.01
EFQ	40	0	2	38	0	0	0	0%	0.9
IQE	48	48	0	0	0	0	0	100%	0.1
ΠE	48	0	0	48	0	0	0	0%	0.02
ΣE	68	68	0	0	0	0	0	100%	0.1
ΣF	32	0	0	32	0	0	0	0%	0.02
ΣI	18	8	8	2	0	0	0	44%	9.2
VAR	68	68	0	0	0	0	0	100%	0.04

Table 2: Rule Statistics for JSeM 720. The six outcome columns (Suc., Fail, Dep., Loop, Time, Pend.) are mutually exclusive and sum to Att.

outermost `deduce'` call; below it, ΣI and EFQ are tried as alternative root rules. Green subtrees reach leaves by forward reasoning (ΣE followed by VAR); the red subtrees mostly hit the depth bound.

3.3 Flame Graph

The Flame Graph adapts a now-standard profiling visualization (Gregg, 2016) to the proof search call stack: each cell's width encodes the total wall-clock time spent in the subtree rooted at that node, and the vertical axis encodes recursion depth. Wide cells are temporal hotspots. The view condenses the same structure shown in Figure 3 into a time-weighted form, so a developer can identify which subtree dominated wall-clock time at a single glance.

3.4 Rule Statistics

The Rule Statistics view groups every search-tree node by the rule that produced it and reports, per rule, the number of attempts, the breakdown of outcomes (success, failure, depth-bound exhaustion, loop avoidance, time limit, pending), the success rate, and the total time spent. Columns are sortable, so a developer can quickly see which rule dominates cost and which rule fails for which reason. Table 2 reports the aggregation for JSeM 720: forward-reasoning rules (VAR, CON, ΣE, IQE) succeed uniformly and contribute negligible time; ΠE, EFQ, and ΣF are attempted many times but never succeed, mostly terminating at the depth bound; backward ΣI accounts for the dominant share of total wall-clock time despite a 44% success rate.

3.5 Failure Analysis

The Failure Analysis panel groups failure events (DEDUCEFAILED, DEPTHEXCEEDED, AVOIDLOOP, TIMELIMIT) by category and, within each category, counts repeated failures on the same goal. Figure 7 (Appendix C) shows the aggregation for JSeM 720: failures are overwhelmingly DEPTHEXCEEDED (over 90% of all failure events), with the remainder DEDUCEFAILED, and the same goal recurs many times within DEPTHEXCEEDED. This indicates that the search repeatedly hits the depth bound on closely related sub-problems—a pattern that is hard to see from raw textual logs but immediately apparent here.

4 Evaluation

We evaluate the tool from two angles: §4.1 demonstrates what is revealed about the JSeM 720 running example, and §4.2 verifies that what is visible faithfully reflects *wani*'s actual search log.

4.1 Diagnostic Case Study: JSeM 720

The four panels together expose the following facts about *wani*'s behavior on JSeM 720, none of which are directly visible in the textual debug log of Figure 2:

Where time concentrates. Table 2 and Figure 4 together show that ΣI dominates total wall-clock time (9.2 ms out of approximately 10.4 ms, about 89%); all other rules together account for about 1.2 ms.

Direct guidance for ML acceleration. *wani* makes a substantial number of attempts at ΠE, ΣF, and EFQ (48, 32, and 40 respectively in our run), even though these attempts almost always terminate by exhausting the depth bound and do not contribute to the final proof. Our tool identifies these three rules as the primary targets for *Neural Wani*'s learned ranker to prune, directly guiding the development of more efficient neuro-symbolic inference for NLI.

Where failures cluster. Figure 7 (Appendix C) shows that failure events cluster on a small set of subgoals, mostly concerning the `entity` type, with the same goal recurring across many `depth_exceeded` terminations. This pattern points to a specific subgoal type where prioritization by a learned ranker could yield the largest reduction in wasted exploration.

4.2 Structural Faithfulness

wani's proof search exhibits a highly branching structure: it explores many alternative branches per goal, fails in heterogeneous ways (depth bound, loop avoidance, time limit, exhaustion of all rules), and may revisit closely related subgoals across

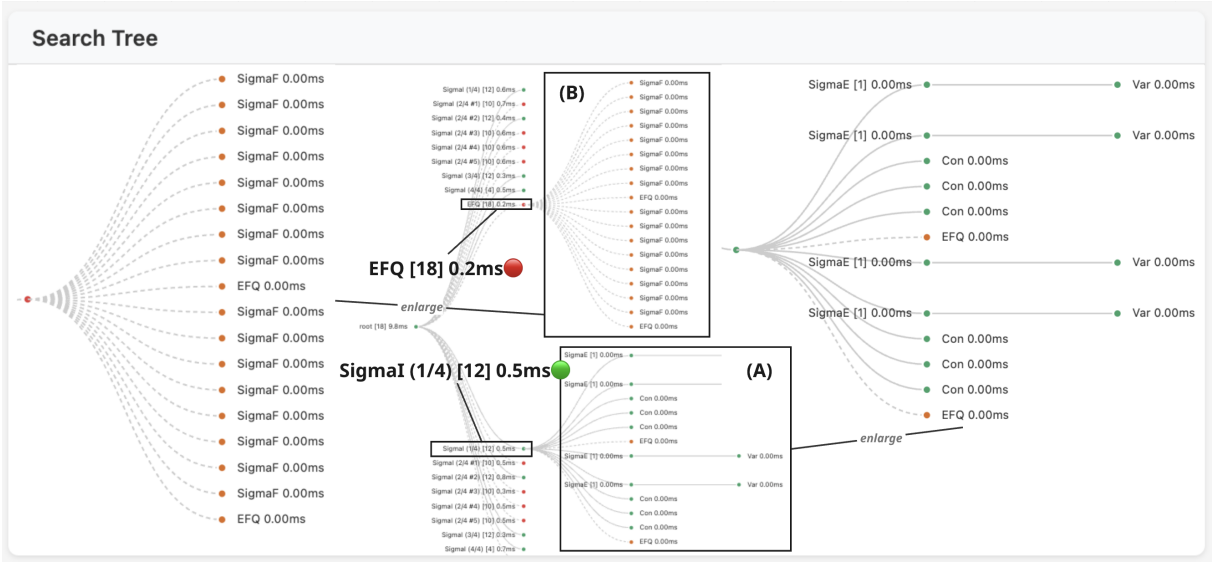

Figure 3: Search Tree for JSeM 720. The central tree shows the direct children of the root, with two enlarged callouts. **(A)** A successful ΣI sub-problem (parent node ΣI (1/4) [12] 0.5 ms in the central tree), expanded to show its forward-reasoning leaves: ΣE followed by VAR, or CON. **(B)** The stalled EFQ sub-problem (parent node EFQ [18] 0.2 ms in the central tree), where its ΣF and EFQ subgoals all exhaust the depth bound. The contrast between (A) and (B) is exactly the kind of signal a learned ranker could exploit. Each node is one deduce' invocation; colors indicate outcome (green: successful discharge; orange: depth-bound exhaustion; red: failure after all applicable rules).

iterative-deepening rounds. Because the Search Tree is reconstructed post-hoc from a flat event log rather than from explicit parent pointers, a reconstruction error would silently distort every downstream view; it must therefore recover this structure faithfully. We verify that it does by checking three structural invariants of the underlying event sequence: (i) *goal pairing*, where every GOALSTART has exactly one matching GOALEND, ensuring no dangling nodes; (ii) *outcome uniqueness*, meaning each goal carries at most one terminal outcome event, ruling out double-counting in the Rule Statistics aggregation; and (iii) *depth invariant*, whereby every GOALSTART at depth $d > 1$ has an active predecessor at depth $d - 1$, formalizing the strict depth-first property required for inferring parent–child structure from recursion depth. We apply the checks to two contrasting problems, JSeM 720 (indirect passive; 336 goals, 1414 events) and JSeM 699 (distributive predication; 1680 goals, 6120 events; query in Appendix A), and find that all three invariants hold without violation in both cases (Table 3). The structural invariants verify that the post-hoc reconstruction logic is sound at problem scales differing by nearly a factor of five and across distinct linguistic phenomena. The check is implemented as a Python script and will be included in the public release, so the same procedure can be applied to any other proof search query.

Check	JSeM 720	JSeM 699
goal pairing	336 / 336 pass	1680 / 1680 pass
outcome uniqueness	336 / 336 pass	1680 / 1680 pass
depth invariant	336 / 336 pass	1680 / 1680 pass

Table 3: Structural faithfulness checks on two JSeM problems of contrasting scale and linguistic phenomena: JSeM 720 (indirect passive; 1414 events / 336 goals) and JSeM 699 (distributive predication; 6120 events / 1680 goals). All three invariants hold without violation in both cases.

5 Related Work

LLM-assisted theorem proving. The line of work introduced in Section 1, LeanDojo, Lean Copilot (Song et al., 2025), Baldur (First et al., 2023), Proverbot9001 (Sanchez-Stern et al., 2020), COPRA (Thakur et al., 2023), Graph2Tac, and, in the NLI context, *Neural Wani*, all aim at making proof search faster by integrating learned components. Our work is complementary: rather than proposing a learned strategy, we provide observability into proof search, measuring where time is spent and exposing failure patterns, thereby enabling and evaluating such interventions.

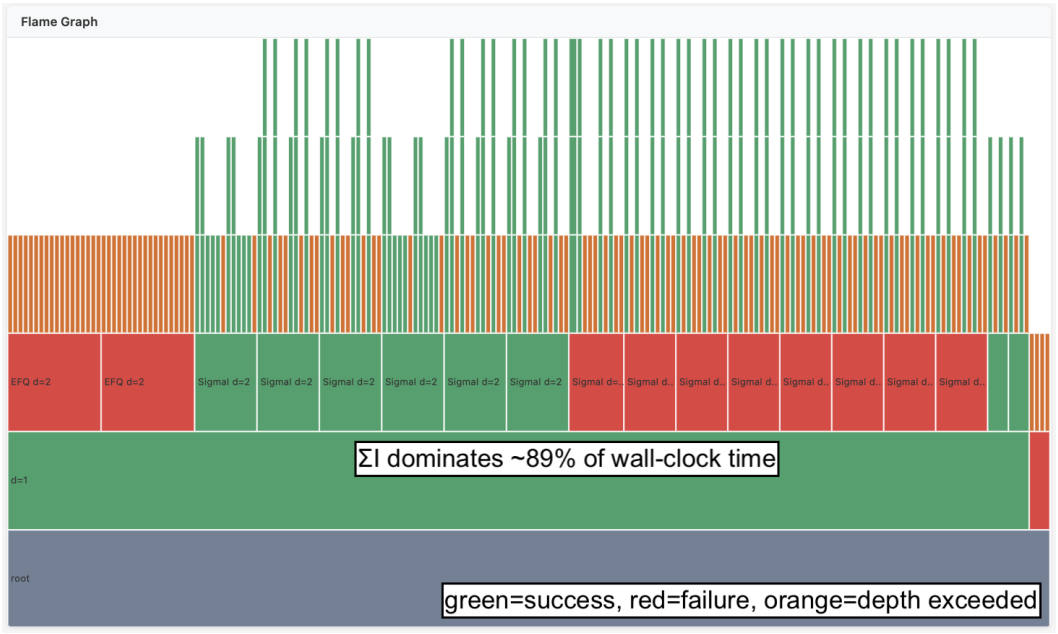

Figure 4: Flame Graph panel for JSeM 720. Cell widths are proportional to subtree wall-clock duration. Colors: green = success, red = failure, orange = depth-bound exhaustion.

Logic-based NLI. Beyond DTS, NLI has also been pursued with higher-order logic and off-the-shelf provers, notably ccg2lambda, which uses Coq to prove entailments between compositionally derived representations. Such systems share the observability gap our tool addresses and are natural portability targets.

Proof-state visualization in interactive provers. Tools such as CoqIDE, vscoq, and Lean's Infoview render the constructed proof and the current tactic state for the developer. They focus on the proof being built, not on the search that produced it: the structure of explored alternatives, the time distribution across rules, and recurring failure patterns, which are precisely the aspects our tool makes observable.

Program profiling. The Flame Graph has become a standard visualization in general software performance analysis. Our work adapts this idea to proof search, where the call structure is not explicit and must be reconstructed from logical inference traces.

6 Conclusion

We presented a profiling and visualization tool that provides observability into proof search in formal natural language inference. The tool exposes *wani*'s search behavior through four coordinated views—Search Tree, Flame Graph, Rule Statistics, and Failure Analysis—reconstructed post-hoc from a structured event log, and is released as open-source software with a Docker image.

The case study on JSeM demonstrates that the tool can surface actionable signals invisible in raw textual logs: which rules dominate wall-clock time, which rules are repeatedly attempted without contributing to the proof, and which subgoal types concentrate failure events. These are precisely the signals that a learned rule ranker needs in order to decide what to prune.

Beyond diagnostics, the structured event log doubles as a training-data generator for ML-acceleration efforts such as *Neural Wani*. Prior to this tool, search paths that led to failure were lost; the profiler now makes them fully recoverable, enabling extraction of both positive and negative training examples for a learned ranker.

Future work proceeds along three directions: (i) applying the tool to corpus-level analysis across JSeM to identify systematic bottleneck patterns beyond individual problems; (ii) porting the profiling framework to other proof-search-based NLI systems by emitting the same event schema, beginning with a Coq-based pipeline such as ccg2lambda and extending toward Lean-based provers; and (iii) directly using the collected positive and negative signals to train rule-selection heuristics within *Neural Wani*.

Limitations

Scope. The current implementation instruments *wani* specifically, but by design the prover-specific part is confined to a single component. *wani*'s only modification is one optional field in its settings record that emits timestamped, structured events. Everything downstream is independent of *wani* and operates on the event sequence alone: the fourteen-kind event schema (Table 1), the post-hoc reconstruction of the four views, and the Python invariant checker of §4.2. Porting the tool to another proof-search-based NLI system therefore reduces to emitting the same schema from that system's search procedure; the reconstruction and visualization layers are reused unchanged. We have not yet performed such a port, and exposing the schema as a shared, prover-independent log format that other systems can target is left as future work.

Rendering scalability. The Search Tree auto-collapses nodes at depth four or deeper to remain interactive at the trace sizes we have measured (up to a few thousand events). For substantially larger proofs, scaling the rendering or providing aggregate-only views is left as future work.

Timing precision. Per-goal durations are computed from the timestamps of structured event records and therefore include the (small) overhead of the logging calls themselves. Reported durations are appropriate for identifying hotspots and relative comparisons, but not for microsecond-level benchmarking.

Iterative-deepening semantics. The event log accumulates events across all iterative-deepening rounds. Users analyzing successful proofs might find the inclusion of events from previous iterative-deepening rounds counterintuitive; however, round boundaries are recoverable from ITERDEEPEN events.

Evaluation scope and ML integration. Our evaluation focuses on a small number of case studies. Corpus-level aggregate evaluation across many JSeM problems is left for future work. We have not conducted a formal user study, as the target user group (developers of DTT-based NLI provers) is highly specialized and currently limited in size, making large-scale quantitative evaluation infeasible at present; qualitative evaluation is left for future work. Finally, this paper presents the observability infrastructure required for ML-acceleration efforts; we do not ourselves implement or evaluate a learned heuristic.

Acknowledgments

This work was supported in part by JST CREST Grant Number JPMJCR2565 and JSPS KAKENHI Grant Number JP23H03452, Japan.

References

Daisuke Bekki and Koji Mineshima. 2017. *Context-Passing and Underspecification in Dependent Type Semantics*, pages 11–41. Springer International Publishing, Cham.

Lasse Blaauwbroek, Miroslav Olšák, Jason Rute, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. 2024. Graph2tac: Online representation learning of formal math concepts. In *International Conference on Machine Learning*.

Hinari Daido and Daisuke Bekki. 2020. Development of an automated theorem prover for the fragment of DTS. In *Proceedings of the 17th International Workshop on Logic and Engineering of Natural Language Semantics (LENLS17)*.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1229–1241, New York, NY, USA. Association for Computing Machinery.

Brendan Gregg. 2016. The flame graph. *Commun. ACM*, 59(6):48–57.

Ai Kawazoe, Ribeka Tanaka, Koji Mineshima, and Daisuke Bekki. 2015. An inference problem set for evaluating semantic theories and semantic processing systems for japanese. In *Proceedings of the Twelfth International Workshop on Logic and Engineering of Natural Language Semantics (LENLS12)*, pages 67–73, Tokyo-Yokohama, Japan.

Per Martin-Löf. 1984. *Intuitionistic Type Theory Vol. 1*. Bibliopolis.

Pascual Martínez-Gómez, Koji Mineshima, Yusuke Miyao, and Daisuke Bekki. 2016. ccg2lambda: A compositional semantics system. In *Proceedings of ACL-2016 System Demonstrations*, pages 85–90, Berlin, Germany. Association for Computational Linguistics.

Koji Mineshima, Pascual Martínez-Gómez, Yusuke Miyao, and Daisuke Bekki. 2015. Higher-order logical inference with compositional semantics. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2055–2061, Lisbon, Portugal. Association for Computational Linguistics.

Nanako Miyagawa, Hinari Daido, and Daisuke Bekki. 2026. Neural Wani: Izon gata riron no tame no jidou teiri shoumeiki wani no kousokuka ni mukete (trans. neural Wani: Toward speeding up the automated theorem prover Wani for dependent type theory). In *Proceedings of the 32nd Annual Meeting of the Association for Natural Language Processing*, pages 2930–2934. In Japanese.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg. Springer-Verlag.

Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2020, page 1–10, New York, NY, USA. Association for Computing Machinery.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2025. Lean copilot: Large language models as copilots for theorem proving in lean. *Preprint*, arXiv:2404.12534.

Amitayush Thakur, George D. Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2023. An in-context learning agent for formal theorem-proving.

The Coq Development Team. 2024. *The Coq Proof Assistant Reference Manual*.

Asa Tomita, Mai Matsubara, Hinari Daido, and Daisuke Bekki. 2025. Natural language inference with CCG parser and automated theorem prover for DTS. In *Proceedings of the Second Workshop on the Bridges and Gaps between Formal and Computational Linguistics (BriGap-2)*, pages 1–7, Düsseldorf, Germany. Association for Computational Linguistics.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 21573–21612. Curran Associates, Inc.

A Proof Search Query for JSeM 699

Figure 5 shows the proof search query for JSeM 699, used as a contrasting example in Section 4.2. Compared with the running example JSeM 720 (Figure 1), it differs in both linguistic phenomenon (distributive predication versus indirect passive) and search scale (1680 versus 336 goals).

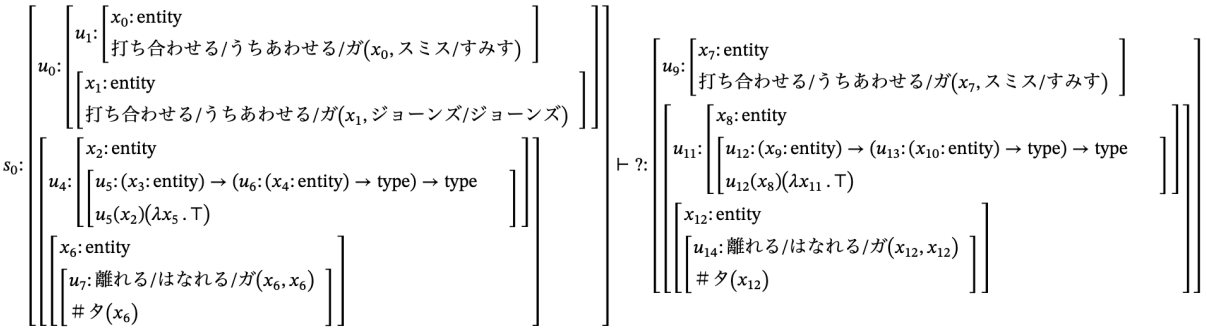

Figure 5: Proof search query for JSeM 699.

B Proof-Structure Dependency Pattern for JSeM 720

For JSeM 720, the search log records that *wani* returns two proof terms inhabiting the goal G_h (`deduced 2 trees` at depth 1). At the root, ΣI is accepted. At depth 2, the RULE_ACCEPT events break down as ΠE 24, EFQ 18, VAR 16, TOPI 2, and ΣF 2. The pattern itself is determined by the type dependencies in the goal. $G_h = \Sigma\, x_2{:}\mathsf{entity}.\, \Sigma\, u_3{:}\mathsf{cry}(x_2, \mathsf{hanako}).\, \mathsf{tense}(x_2)$ is inhabited by a triple $\langle a, \langle b, c\rangle\rangle$ with $a : \mathsf{entity}$, $b : \mathsf{cry}(a, \mathsf{hanako})$, and $c : \mathsf{tense}(a)$. Inside s_0, the only choice of a for which both the dependent cry and tense components are well-typed is x_0: the only proof of $\mathsf{cry}(\cdot, \mathsf{hanako})$ available in s_0 has type $\mathsf{cry}(x_0, \mathsf{hanako})$, and the only tense term in s_0 has type $\mathsf{tense}(x_0)$. Once $a = x_0$ is fixed, u_0 and $\mathsf{tense}(x_0)$ are forced as the components for b and c. The components specific to the adversative-passive marking (x_1, u_1) cannot participate in this well-typed assignment.

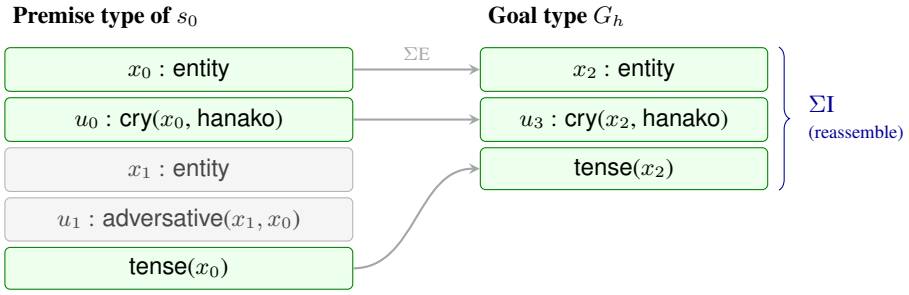

Figure 6: Dependency pattern for any inhabitant of G_h: ΣI at the root combines components projected from the premise variable s_0 via ΣE (each arrow stands for a chain of projections walking down s_0's nested Σ structure). The components specific to the adversative-passive marking (x_1, u_1, in gray) cannot contribute on type-coherence grounds. Predicate names are romanized from the original DTS output of Figure 1.

C Failure Analysis Panel for JSeM 720

Figure 7: Failure Analysis panel for JSeM 720.