From Execution to Exploration: Bridging the Usability Gap in Formal Natural Language Inference

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

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

Abstract

Linguistically oriented formal NLI systems ensure the validity and transparency of inference. However, the combinatorial explosion of candidates, which we term the *branching problem*, imposes prohibitive computational overhead and a heavy cognitive burden on grammar developers. We argue that a central cause is a mismatch between the exhaustive execution paradigm and the actual workflow of grammar developers. To overcome this barrier, we propose restructuring the development workflow from exhaustive execution to interactive exploration driven by developer decisions. We realize this shift in *Express*, a web-based interactive development environment for *lightblue*, a Japanese automated inference system built upon Combinatory Categorial Grammar and Dependent Type Semantics. *Express* transforms branches at each stage of parsing, type checking, and proof search into explicitly selectable units, transferring control over the reasoning process to the developer. Our evaluation shows that this paradigm shift effectively reduces unnecessary computation and cognitive burden during grammar development: in a user study, we observed a 96% reduction in explored paths and an improvement in the task success rate from 25% to 100%. Furthermore, a case study demonstrates a roughly 12× reduction in debugging turnaround time.

1 Introduction

Linguistically oriented approaches to natural language inference (NLI), ranging from automated systems (Bos, 2008; Mineshima et al., 2015; Abzianidze, 2017; Hu et al., 2019) to proof-assistant-based approaches (Chatzikyriakidis and Luo, 2014), ground inference in logical forms and theorem proving, ensuring both validity and transparency. Yet the grammar developers who build and maintain these systems struggle to use them effectively in everyday development.

The system targeted in this work, *lightblue* (Tomita et al., 2025), is a formal NLI system based on CCG and DTS (see §2.1 for details). *lightblue* serves two complementary use cases: as an automated inference engine whose outputs are used for NLI experiments and treebank construction (Tomita et al., 2024), and as a platform for grammar development, where developers trace through the inference pipeline to verify that their theoretical assumptions yield the intended linguistic predictions. It is this second use case—a feedback loop between linguistic theory refinement and computational verification—that *Express* is designed to support.

However, the development and verification of such systems pose unique challenges. Inference in *lightblue* consists of multiple stages (syntactic/semantic parsing, type checking, and proof search), each of which retains multiple theoretically admissible candidates as it proceeds. We refer to the resulting proliferation of candidates as the *branching problem*. Unlike conventional NLI systems where sentence-level ambiguities are resolved independently, *lightblue*'s DTS-based architecture performs type checking sequentially: each syntactic structure may yield multiple type checking outcomes due to alternative anaphora/presupposition resolutions, and each such outcome serves as the context for type checking the subsequent sentence. As a result, ambiguities compound across sentences, causing the total number of branches to grow exponentially even for a typical NLI problem with just two or three input sentences.

Under the conventional exhaustive execution approach, computation is uniformly applied to all branches, including those irrelevant to the developer's interests. In practice, developers must manually sift through voluminous output to locate a single path of interest among hundreds of candidates, a cognitively demanding and error-prone process that constitutes a major bottleneck in everyday

Third Workshop on the Bridges and Gaps between Formal and Computational Linguistics, pages 1–11

grammar development.

While interactive tools exist for grammar engineering and NLP pipeline inspection—XLE (Crouch et al., 2011) for LFG, INCEpTION (Klie et al., 2018) for annotation, AllenNLP Demo (Gardner et al., 2018) for visualization—they target a single processing phase and do not track chains of ambiguity across fundamentally different stages such as parsing, type checking, and theorem proving.

More closely related is the use of proof assistants for NLI: Chatzikyriakidis and Luo (2014) use Coq's interactive capabilities to verify natural language inferences. However, proof assistants operate within a single proof goal, whereas the branching problem targeted here arises upstream, at the level of selecting which syntactic structures and type checking outcomes to pass to the prover. Our approach makes this upstream disambiguation itself interactive, a level of control that proof assistants do not address.

To our knowledge, no integrated interactive development environment has been proposed for formal NLI systems that provides unified visualization and control of ambiguity across parsing, type checking, and proof search.

We argue that the root cause of this usability issue lies not in computational complexity or interface design, but in a fundamental gap between the exhaustive execution paradigm (which computes everything before presenting results) and the actual workflow of grammar developers (who naturally focus on specific branches, verify them incrementally, and apply their linguistic intuition to select among candidates).

To bridge this gap, we propose restructuring the development workflow from execution to exploration: instead of inspecting exhaustively computed results, developers navigate the inference pipeline interactively, guided by their linguistic judgment. We realize this shift in *Express*, a web-based interactive development environment for *lightblue* that makes branches explicitly selectable units and transfers control over the reasoning process back to the developer. Implemented in Haskell—the same language as *lightblue*—using the Yesod framework (Snoyman, 2015), *Express* directly reuses *lightblue*'s internal data structures.

Our contributions are threefold: (1) An interactive execution model for *lightblue* in which disambiguation advances incrementally through user interaction; (2) *Express*, a web-based environment with unified branch visualization across parsing, type checking, and proof search; and (3) Empirical validation through a user study demonstrating substantial improvements in both task success rate and completion time.

Our evaluation shows that this shift effectively avoids unnecessary computation and reduces cognitive burden (§5).

2 The Branching Problem

This section formally defines how branches proliferate across *lightblue*'s multi-stage inference pipeline (syntactic/semantic parsing, type checking, and proof search), and argues that the resulting computational overhead and cognitive burden stem from a paradigm mismatch.

2.1 Multi-Stage Inference in Formal NLI

lightblue is an automated inference system that integrates Combinatory Categorial Grammar (CCG; Steedman, 2000) with Dependent Type Semantics (DTS; Bekki and Mineshima, 2017). Its inference pipeline consists of three sequential stages (Tomita et al., 2025):

Parsing and semantic composition. CCG-based syntactic parsing is applied to input sentences, yielding DTS-based semantic representations. Since CCG parsing employs CYK chart parsing, multiple grammatically admissible syntactic structures may arise from various combinations of syntactic types or alternative category assignments to lexical items.

Type checking and anaphora resolution. Type checking under Dependent Type Theory (Martin-Löf, 1984) is applied to each resulting semantic representation. When a representation contains underspecified terms (e.g., pronouns and other presupposition triggers that require resolution), anaphora resolution is simultaneously carried out through proof search; this process may yield multiple proof terms, causing the type checking results to branch. For multi-sentence inputs, the semantic representation of each preceding sentence serves as the context for type checking the subsequent one, causing the combinations of syntactic structures and type checking outcomes to grow exponentially in the number of sentences.

Proof search. The theorem prover *wani* (Daido and Bekki, 2020) determines whether a proof term can be constructed for entailment (the hypothesis sentence) or contradiction (its negation) to output YES, NO, or UNKNOWN labels.

2.2 Formal Definition of the Branching Problem

Given k input sentences ($k-1$ premises and one hypothesis), let i denote the maximum number of syntactic structures retained per sentence and j the maximum number of type checking outcomes per structure. The total number of proof search queries grows exponentially in k, reaching:

$$2 \cdot (i \cdot j)^k \tag{1}$$

In practice, parsing may yield multiple syntactic structures from syntactic type ambiguity, but only the top-n are retained; similarly, each structure may contain underspecified terms whose anaphora resolution and presupposition binding admit multiple proof terms, but only the top-m type checking outcomes are kept. Thus i and j in Equation (1) denote these bounded counts (e.g. $i = 4$, $j = 4$ in §5), yet the branching problem persists: because type checking proceeds sequentially, each sentence's result serving as context for the next, the total still grows as $(i \cdot j)^k$, exponentially in the number of sentences k.

We term this the *branching problem* (Figure 1). The factor of 2 reflects proof search for both entailment and contradiction.

Under the conventional exhaustive execution approach, all candidates are enumerated and processed uniformly. This leads to two critical consequences:

Computational overhead. In this paradigm, all potential paths are computed indiscriminately, including those irrelevant to the developer's linguistic intent. Even for typical NLI problems consisting of only two or three sentences, the number of branches increases exponentially with respect to k, requiring substantial computational resources to isolate the inference result for a specific path of interest.

Cognitive burden. The exhaustive approach intermingles results from all disambiguation paths, often flooding the developer with outputs from spurious or unintended branches. Developers are forced to manually sift through voluminous, interleaved data to locate the single reasoning path they wish to verify—a cognitively demanding and error-prone process that stifles rapid, iterative development and debugging.

2.3 The Root Cause: A Paradigm Mismatch

The branching problem reflects a fundamental mismatch between the system's "compute-then-output" execution paradigm and the actual workflow of grammar developers, who rarely need all inference results simultaneously.

For instance, a developer may wish to isolate a specific syntactic structure for a single sentence to observe its downstream effects on type checking and proof search. Alternatively, they may need to verify whether a recent modification to a grammar correctly reflects in the intermediate syntactic structures without running the entire pipeline. In these scenarios, what the developers actually require is selective verification, step-by-step control, and inspection of intermediate stages.

The exhaustive execution paradigm fundamentally fails to support this selective, incremental workflow. Because results are exclusively presented after the entire computation completes, the system operates as a monolithic process. The developer cannot intervene at intermediate stages of inference to prune unintended branches, directly causing the computational waste and cognitive overload discussed previously.

3 From Execution to Exploration

The mismatch identified in the previous section is rooted in the execution paradigm itself. Importantly, the shift we propose does not alter *lightblue*'s automated inference engine; rather, it restructures the development workflow so that developers can navigate the inference pipeline interactively, as detailed below.

Under the exploration paradigm we propose, developers navigate the inference pipeline stage by stage: at each stage, the system presents the available branch candidates, and the developer selects one before computation proceeds. Because the developer explicitly selects one candidate at each branching point, the number of active paths never grows to $2 \cdot (i \cdot j)^k$ as under exhaustive execution, but remains manageable at all times, simultaneously avoiding unnecessary computation and reducing cognitive burden.

This shift is not merely an interface improvement: a post-hoc filtering approach that layers a

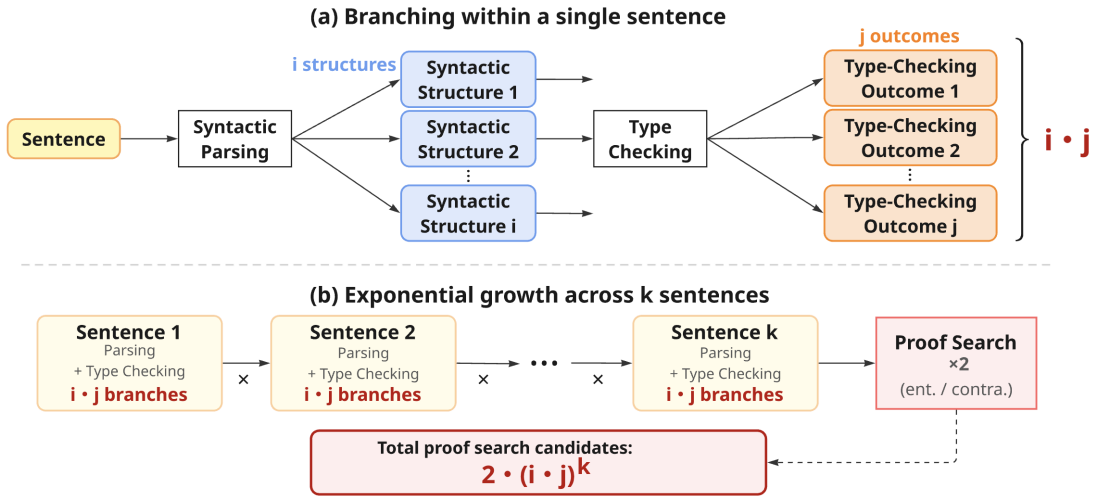

Figure 1: Branching structure of inference in *lightblue*. Each sentence retains i top-scoring syntactic structures and j type checking outcomes per structure, resulting in up to $2 \cdot (i \cdot j)^k$ proof search queries for k input sentences.

tree-structured UI over exhaustively computed results would reduce cognitive burden but would entirely fail to address computational overhead. Indeed, when relevant branches are not reached within practical time limits (as with Problem B in §5.1), there are no results to filter. The exploration paradigm resolves this by integrating developer decisions into the execution itself: exploiting Haskell's lazy evaluation (§4.3), computation proceeds on demand only along the path the developer selects, so that results can be obtained even when exhaustive enumeration is infeasible.

4 Express

Express realizes the exploration paradigm as a web-based interactive development environment for *lightblue*, enabling step-by-step control over branch selection across the entire inference pipeline.[1]

4.1 Design Principle

The design of *Express* is guided by the principle that navigating the inference pipeline should be an interactive, user-driven process rather than a monolithic batch computation. In this architecture, branches are treated as explicitly selectable units that the developer can inspect and operate on. Specifically, syntactic structures at the parsing stage and type checking outcomes at the semantic stage are presented as distinct candidates for the developer to evaluate. While the current implementation directly interfaces with *lightblue*'s internal functions, the exploration logic itself is not inherently system-specific. Abstracting the parser and type checker interaction into a Haskell type class, with *lightblue* as one instance, is a planned extension to support other formal NLI systems.

Based on this principle, *Express* integrates three features aligned with the practical debugging workflow: (1) **Investigate:** localized substring analysis; (2) **Explore:** interactive branch selection for full-sentence inference; and (3) **Share:** proof search query export for cross-developer collaboration. Each is detailed in the following subsections.

4.2 Investigate: Substring Analysis

The first step in diagnosing unexpected behavior is to localize the problem to a specific substring. *Express* builds on *lightblue*'s CYK chart parsing—which inherently retains syntactic structures for all intermediate constituents—allowing the developer to select any arbitrary span of the input sentence. The system instantly displays the parsing candidates and their DTS semantic representations for that span (Figure 2), enabling rapid localized diagnosis without re-running the full pipeline.

4.3 Explore: Interactive Branch Selection

Building on this local analysis, the developer must then run full-sentence inference to verify how local modifications affect the final outcome. Under exhaustive execution, the developer would be forced

[1] Source code: https://github.com/kohapizza/lightblue (*Express* is integrated into *lightblue*). A demonstration and usage instructions are available at https://kohapizza.github.io/Express-Demo-Website/.

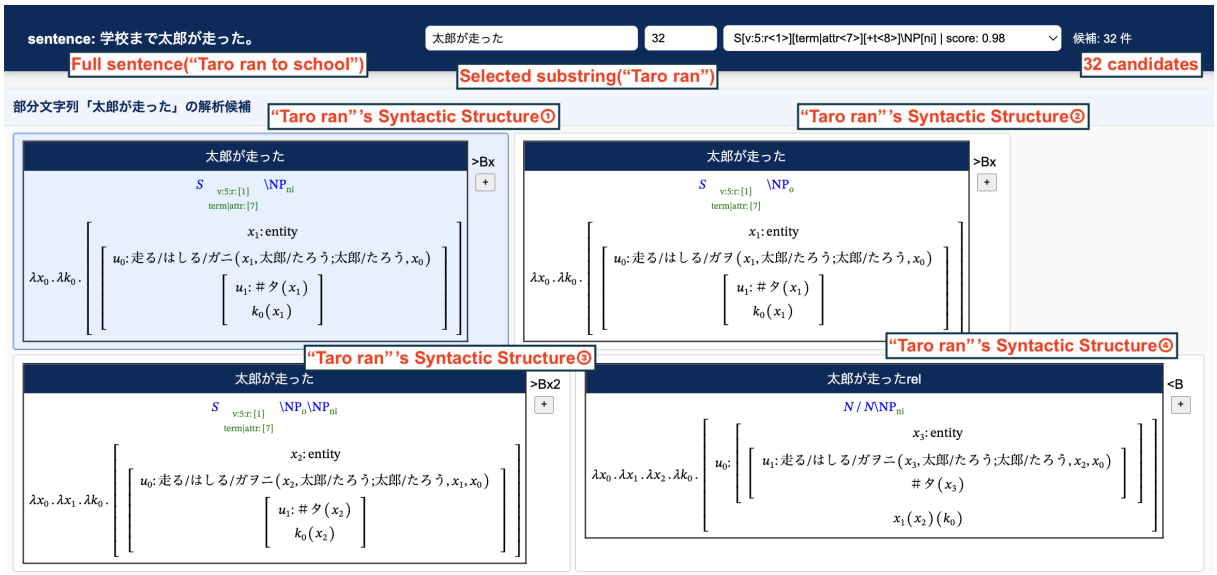

Figure 2: Substring analysis in *Express*. From the full sentence *Taro ran to school*, the developer selects the substring *Taro ran*. *Express* instantly displays all parsing candidates retained in the chart for the selected substring, each showing the CCG syntactic type and DTS semantic representation.

to wade through up to $2 \cdot (i \cdot j)^k$ paths to locate the relevant one.

Express eliminates this bottleneck by decomposing the inference process into two sequential selection points: (1) *syntactic structure selection:* Multiple parsing candidates generated by CCG are displayed as parallel, collapsible cards. The developer selects a single structure to proceed to type checking. (2) *Type checking outcome selection:* Multiple proof terms arising from alternative anaphora/presupposition resolutions are presented as distinct branches. The developer selects one specific outcome to proceed to proof search. Because the developer selects exactly one candidate at each stage, the number of active paths never grows exponentially (Figure 3 and Figure 4).

To ensure responsiveness, *Express* computes inference results on demand via lazy streams. Parsing results are maintained as per-sentence streams, and type checking outcomes are stored in a map keyed by sentence and node indices; changing a branch invalidates only downstream computations. Proof search results are cached by query content, so that different paths converging on the same query share cached results, minimizing redundant computation.

4.4 Share: Proof Search Query Export

When interactive exploration reveals a failure during the proof search phase, the grammar developer must communicate the failure conditions to the theorem prover’s developer. Historically, reproducing specific inference contexts required manually reconstructing the formal proof search query from raw system outputs, a tedious, error-prone process often consuming upwards of an hour for complex semantic representations.

Express directly addresses this friction by providing a feature to export the interactively formulated proof search query into a format natively loadable in *wani*. The theorem prover developer can then instantly load the exported query to accurately reproduce the failure context. This direct handoff accelerates the debugging cycle from initial diagnosis to collaborative resolution (Figure 5).

5 Evaluation

We now examine whether the shift from execution to exploration, as realized in *Express*, reduces unnecessary computation and cognitive burden caused by the branching problem (§2).

5.1 Search Space Reduction

We quantitatively evaluated the search space reduction using two NLI problems that represent typical scenarios grammar developers face: Problem A, which *lightblue* solves correctly but where exhaustive execution is slow due to branching; and Problem B, which is theoretically solvable but not handled successfully by the current theorem prover. Both problems consist of two input sentences ($k = 2$), with the system configured to retain up to 4 syntactic structures per sentence ($i = 4$; see

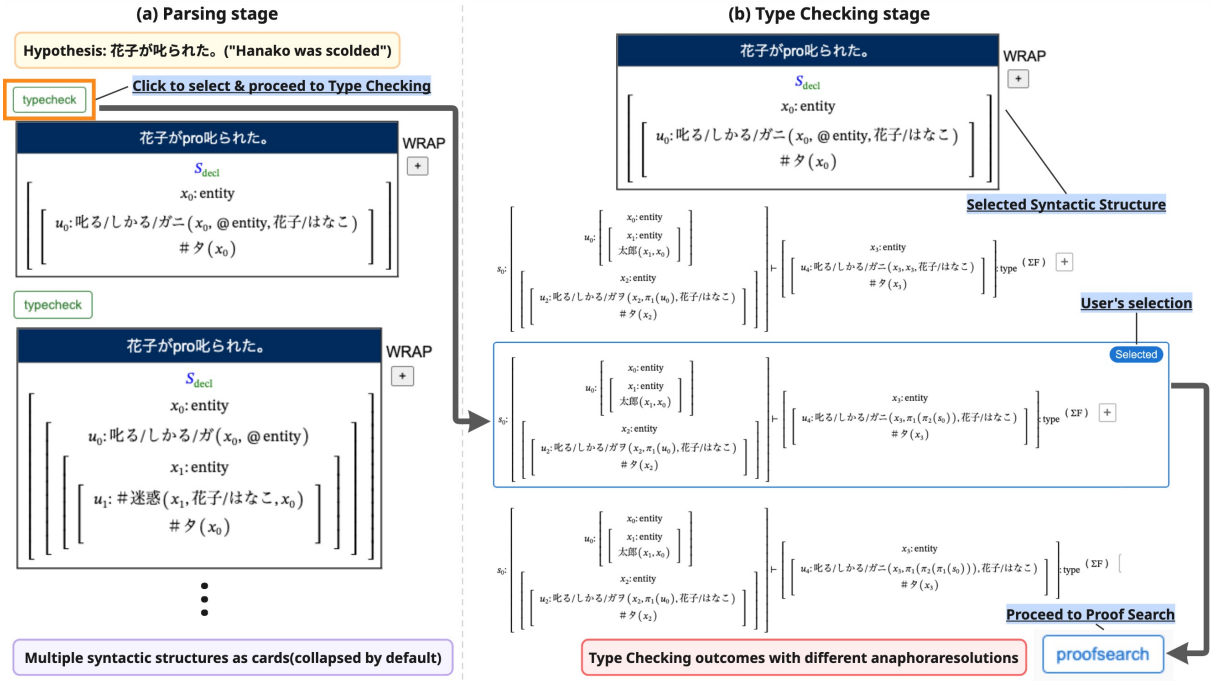

Figure 3: Interactive branch selection in *Express* for the hypothesis *Hanako was scolded*. (a) Multiple syntactic structures are displayed as collapsible cards (UI panels); the developer selects one by clicking `typecheck`. (b) The resulting type checking outcomes, each corresponding to a different anaphora resolution, are displayed; the developer selects one before proceeding to proof search.

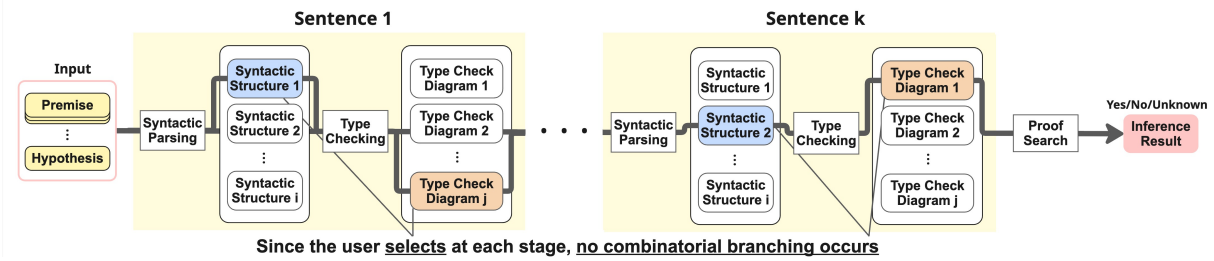

Figure 4: Operational flow of *Express*. The developer incrementally selects syntactic structures and type checking results at each stage (highlighted path), preventing the exponential branching that arises under exhaustive execution.

	Exhaustive		*Express*	
	Paths	Time	Paths	Time
Problem A	260	>120s	10	12s
Problem B	128	>120s	28	94s

Table 1: Search space comparison. Exhaustive execution was terminated at the 120-second time limit. Paths: number of disambiguation paths computed; Time: backend computation time.

§2.2) and up to 4 type checking outcomes per structure ($j = 4$), producing at most $2 \cdot (4 \cdot 4)^2 = 512$ proof search queries under exhaustive execution.

For Problem A, exhaustive execution generated 260 paths and timed out, whereas *Express* required only 10 paths and completed in 12 seconds, a 96% reduction (Table 1). For Problem B, exhaustive execution had evaluated 128 paths when terminated at the 120-second time limit; the enumeration remained incomplete and did not reach the target proof search query, which exists in the full search space. In contrast, *Express* enabled the developer to reach this query after evaluating only 28 paths and export it, enabling the kind of collaborative debugging illustrated in §5.3. These results show that interactive exploration effectively avoids unnecessary computation: developers can reach diagnostically relevant branches that are theoretically present but practically inaccessible under exhaustive execution.

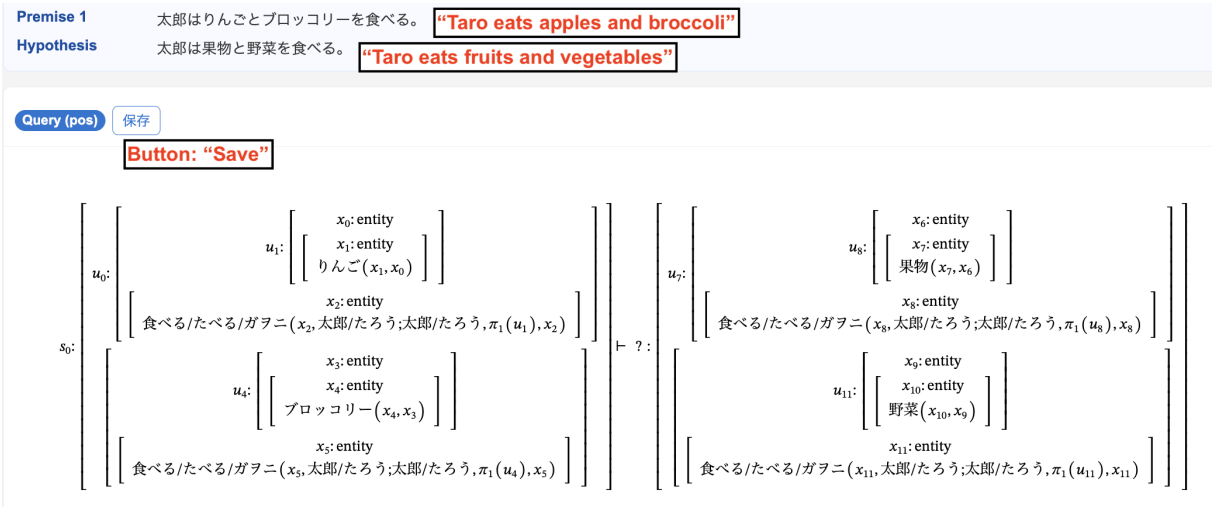

Figure 5: Proof search query export in *Express*. The premise *Taro eats apples and broccoli* and hypothesis *Taro eats fruits and vegetables* are displayed with the corresponding proof search query as a DTS type judgment. The query can be saved as a file directly loadable by the theorem prover *wani*.

5.2 User Study

To examine whether the reduction in search space translates into a reduced cognitive burden for developers, we conducted a within-subjects user study.

Participants and task design. Four researchers with experience in grammar development using *lightblue* participated, representing nearly the entire population of active *lightblue* grammar developers. The task was to identify a specific disambiguation path and locate the corresponding proof search query, using Problems A and B from §5.1. In the *Express* condition, participants used *Express*; in the baseline, they navigated a static HTML file containing all disambiguation paths from exhaustive execution. To control for order effects and problem difficulty, we employed a counterbalanced design: each participant solved one problem with *Express* and the other with the baseline, varying both the problem assignment and presentation order across participants (Table 2). This design directly isolates and measures the usability of inference process navigation, decoupled from *lightblue*'s underlying inference accuracy.

Results. A notable result is the contrast in task success rates: all four participants successfully identified the target path in the *Express* condition (100%), whereas only one succeeded in the baseline condition (25%).

The mean completion time decreased from 7m 20s to 1m 51s (Table 2). Note that the baseline mean is highly conservative; two participants reached the 10-minute time limit without completing the task, meaning their true completion times would have been longer. Participants' qualitative feedback indicates that the improvement lies not merely in speed, but in reduced cognitive load. In the baseline condition, P2 reported "I couldn't tell where I was in the file, and couldn't return to where I had been looking." In contrast, *Express* provided "an intuitive sense of the path." This suggests that transferring control over the reasoning process to developers enables them to maintain their spatial orientation within the inference space, substantially reducing the cognitive burden imposed by the exhaustive paradigm.

ID	Cond.	Prob.	Time	Ops.	Success
P1	Baseline	A	4m 08s	46	Yes
P1	*Express*	B	1m 26s	8	Yes
P2	Baseline	B	10m 00s	–	No (timeout)
P2	*Express*	A	1m 45s	14	Yes
P3	*Express*	A	2m 55s	16	Yes
P3	Baseline	B	10m 00s	–	No (timeout)
P4	*Express*	B	1m 17s	8	Yes
P4	Baseline	A	5m 12s	–	No
Baseline			7m 20s (avg)	46	1/4 (25%)
Express			1m 51s (avg)	11.5 (avg)	4/4 (100%)

Table 2: User study results. Rows are ordered by presentation sequence within each participant. Prob.: problem from Table 1. Time: task completion time (timeout at 10 min). Ops.: number of operations; omitted for failed trials.

Subjective usability ratings (Table 3) corroborate this interpretation: *Express* received near-maximum scores across all items, with a particularly pronounced advantage in the ease of under-

	Q1	Q2	Q3	Q4
Baseline	1.75 (1.5)	1.25 (1.0)	–	–
Express	4.75 (5.0)	4.75 (5.0)	5.00 (5.0)	4.75 (5.0)

Table 3: Likert scale results: mean (median). Q1: ease of reaching the target path; Q2: ease of understanding the inference process; Q3: workload reduction; Q4: intention to use in practice.

standing the inference process (Q2: median 5.0 vs. 1.0).

5.3 Case Study: Diagnosing Proof Search Failures

Finally, we illustrate how the exploration paradigm fosters not only individual efficiency but also collaborative problem-solving across distinct developer roles.

Given the premise "Taro eats apples and broccoli" and the hypothesis "Taro eats fruits and vegetables," *lightblue* returned an UNKNOWN label instead of the expected YES. This entailment should hold once proof search establishes the lexical relations—that apples are a kind of fruit, and broccoli a kind of vegetable. Diagnosing why it did not requires isolating, among the up to $2 \cdot (i \cdot j)^k$ candidates produced under exhaustive execution, the proof search query at issue. The grammar developer carried out this diagnosis in three steps, each corresponding to one of the features in §4.

Localizing the problem (*Investigate*). The developer first employed substring analysis (§4.2) to verify that the relevant segments of the input sentences were syntactically and semantically parsed as intended, thereby isolating the issue and confirming that it did not originate in the syntactic analysis phase.

Pinpointing the unsolved query (*Explore*). The developer then ran full-sentence inference and, at each branching point, selected the intended syntactic structure and type checking outcome (§4.3). Without sifting through the candidates produced by exhaustive execution, this led to the proof search query that should have returned YES but instead returned UNKNOWN. Since the query remains unsolved despite these intended selections, the problem can be attributed to proof search rather than to any earlier stage.

Reporting it to the prover developer (*Share*). Finally, the developer exported this specific query as a file natively loadable by *wani* (Figure 5) and sent it to the theorem prover's developer. Upon loading the file, the *wani* developer reproduced the exact state and identified the root cause, a search efficiency issue related to equality type introduction, within approximately five minutes.

Under the previous workflow, the *wani* developer would have had to manually transcribe the formal expressions from screenshots into DTT format, a tedious process that alone required roughly an hour for complex semantic representations. *Express*'s export feature entirely eliminated this transcription bottleneck, resulting in a roughly $12\times$ reduction in debugging turnaround time.

This case demonstrates that the exploration paradigm extends beyond individual productivity: by enabling developers to isolate and share specific branches in a reproducible format, *Express* makes cross-stage collaboration practically viable.

6 Conclusion

To address the computational overhead and cognitive burden caused by the branching problem, rooted in a mismatch between the exhaustive execution paradigm and the iterative workflow of grammar developers, we proposed restructuring the development workflow from execution to exploration. *Express*, the system realizing this paradigm, transforms branches into explicitly selectable units, transferring control over the reasoning process to the developer. As confirmed by our evaluation (§5), this shift substantially reduces unnecessary computation and cognitive burden in practice.

While our evaluation is currently limited to *lightblue* and Japanese data, we emphasize that its goal is not broad user generalization, but to assess usability for expert developers engaged in grammar engineering.

A natural concern is whether these gains reflect differences in task familiarity or problem difficulty rather than the tool itself. Our within-subjects, counterbalanced design guards against this: each participant performed one task with *Express* and the other with the baseline, with problem assignment and presentation order counterbalanced across participants (Table 2), controlling for both individual expertise and problem difficulty. The effect is moreover consistent across participants: all four succeeded with *Express*, whereas only one succeeded with the baseline and two trials reached the 10-minute timeout. We nonetheless acknowledge that $N = 4$ limits statistical power. Because

this already constitutes nearly the entire population of active *lightblue* developers, a substantially larger study would require that community to grow, which is itself one of the motivations for building *Express*.

While *Express* currently targets *lightblue*, the branching problem is not a limitation of this particular system but an inherent consequence of preserving syntactic ambiguity through the pipeline and performing anaphora resolution within cross-sentential type checking, a design that faithfully reflects the underlying linguistic theory. This combination is uncommon among current formal NLI systems, where ambiguity is typically resolved at a single stage (see Appendix A.1), and the branching problem accordingly does not arise. The paradigm shift we propose, from exhaustive execution to interactive exploration, is not specific to *lightblue*; it can apply to formal NLI systems whose multi-stage pipelines produce compounding ambiguity. As such systems adopt richer cross-stage linguistic representations, the exploration paradigm is likely to become increasingly relevant.

More broadly, formal NLI systems such as *lightblue* represent a bridge between formal linguistics and computational approaches to language understanding: they ground inference in well-defined linguistic theories, making theoretical assumptions explicit and computationally testable. Such systems also hold promise as platforms for evaluating the reasoning capabilities of neural language models against rigorous formal linguistic standards. However, if the development workflow for the systems that embody this bridge remains impractical, the bridge itself cannot be crossed. By making the development workflow of formal NLI tractable and transparent, *Express* helps ensure that the connection between formal linguistics and NLP is not merely theoretical but practically sustainable.

Acknowledgments

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

References

Lasha Abzianidze. 2017. LangPro: Natural language theorem prover. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 115–120, Copenhagen, Denmark. Association for Computational Linguistics.

Daisuke Bekki and Koji Mineshima. 2017. Context-passing and underspecification in dependent type semantics. In *Modern Perspectives in Type-Theoretical Semantics*, Studies in Linguistics and Philosophy, pages 11–41. Springer.

Johan Bos. 2008. Wide-coverage semantic analysis with Boxer. In *Semantics in Text Processing. STEP 2008 Conference Proceedings*, pages 277–286. College Publications.

Stergios Chatzikyriakidis and Zhaohui Luo. 2014. Natural language inference in Coq. *Journal of Logic, Language and Information*, 23(4):441–480.

Dick Crouch, Mary Dalrymple, Ronald M. Kaplan, Tracy Holloway King, John Maxwell, and Paula Newman. 2011. XLE documentation. Palo Alto Research Center. Available at https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/xle_toc.html.

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)*.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. AllenNLP: A deep semantic natural language processing platform. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 1–6, Melbourne, Australia. Association for Computational Linguistics.

Hai Hu, Qi Chen, and Larry Moss. 2019. Natural language inference with monotonicity. In *Proceedings of the 13th International Conference on Computational Semantics - Short Papers*, pages 8–15, Gothenburg, Sweden. Association for Computational Linguistics.

Jan-Christoph Klie, Michael Bugert, Beto Boullosa, Richard Eckart de Castilho, and Iryna Gurevych. 2018. The INCEpTION platform: Machine-assisted and knowledge-oriented interactive annotation. In *Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations*, pages 5–9, Santa Fe, New Mexico. Association for Computational Linguistics.

Per Martin-Löf. 1984. *Intuitionistic Type Theory*, volume 9. Bibliopolis, Naples.

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.

Michael Snoyman. 2015. *Developing Web Apps with Haskell and Yesod*, 2nd edition. O'Reilly Media.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press.

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.

Asa Tomita, Hitomi Yanaka, and Daisuke Bekki. 2024. Reforging: A method for constructing a linguistically valid Japanese CCG treebank. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 196–207, St. Julian's, Malta. Association for Computational Linguistics.

A Appendix - Supplementary Details

A.1 The Branching Problem in Other Formal NLI Systems

ccg2lambda (Martínez-Gómez et al., 2016) translates CCG derivations into first-order or higher-order logical formulas and delegates inference to an off-the-shelf theorem prover. Because its semantic representations do not involve underspecified terms requiring anaphora resolution during type checking, the exponential branching across sentences that characterizes *lightblue*'s DTS-based pipeline does not arise. LangPro (Abzianidze, 2017) similarly resolves lexical and phrasal ambiguities within a natural logic tableau framework where branching is managed by the prover's own search strategy rather than by an external multi-stage pipeline. The branching problem addressed in this paper is thus particularly acute in systems where type checking and anaphora resolution are interleaved across sentences, as in DTS-based architectures.

A.2 User Study Details

Participant expertise. All four participants are researchers at the authors' institution with experience in grammar development using *lightblue*. Table 4 summarizes their experience and research focus.

ID	Exp.	Research focus
P1	3 yrs	NLI system evaluation using formal test suites
P2	1 yr	Neural-symbolic integration for formal NLI
P3	3 yrs	CCG-based semantic composition for Japanese
P4	2 yrs	Neural acceleration of theorem proving

Table 4: Participant expertise. Exp.: years of experience with *lightblue*.

Likert scale questionnaire. Participants rated each item on a 5-point scale (1: strongly disagree, 5: strongly agree). Q1 and Q2 were administered for both conditions; Q3 and Q4 were administered for *Express* only, as they explicitly compare *Express* against the baseline.

Q1 It was easy to reach the target inference path.

Q2 It was easy to understand the inference process.

Q3 *Express* reduced the workload compared to the conventional approach.

Q4 I would like to use *Express* in actual grammar development work.

A.3 NLI Problems Used in the Evaluation

Table 5 shows the two NLI problems used in §5.1 and §5.2. Both problems consist of one premise and one hypothesis (k=2) and involve entailment relations concerning Japanese passive constructions, where branching arises from the syntactic analysis and anaphora resolution of passive forms.

	Problem A	**Problem B**
Premise	*Taro-ga Hanako-ni nak-are-ta.* (Taro was cried on by Hanako.)	*Taro-ga Hanako-o shikat-ta.* (Taro scolded Hanako.)
Hypothesis	*Hanako-ga nai-ta.* (Hanako cried.)	*Hanako-ga shikar-are-ta.* (Hanako was scolded.)
Label	Yes	Yes

Table 5: NLI problems used in the evaluation (§5).

Problem A involves an indirect passive (*adversative passive*) construction, while Problem B involves entailment from an active to a passive sentence. Under exhaustive execution, Problem A generated 260 disambiguation paths and Problem B generated 128 (Table 1).

Figure 6 shows a portion of the HTML output generated by *lightblue*'s conventional exhaustive execution for Problem A, as used in the baseline condition of the user study. The small scroll bars indicate that the visible area represents only a fraction of the full output.

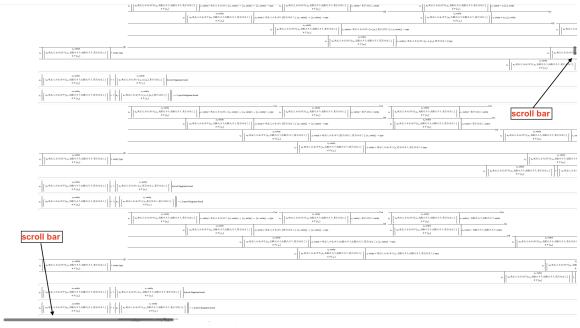

Figure 6: Conventional *lightblue* output for Problem A (3.7 MB HTML file). The developer must manually locate a single path of interest within this voluminous output. Compare with the *Express* interface in Figures 3 and 4.