From Execution to Exploration:

Bridging the Usability Gap in Formal Natural Language Inference

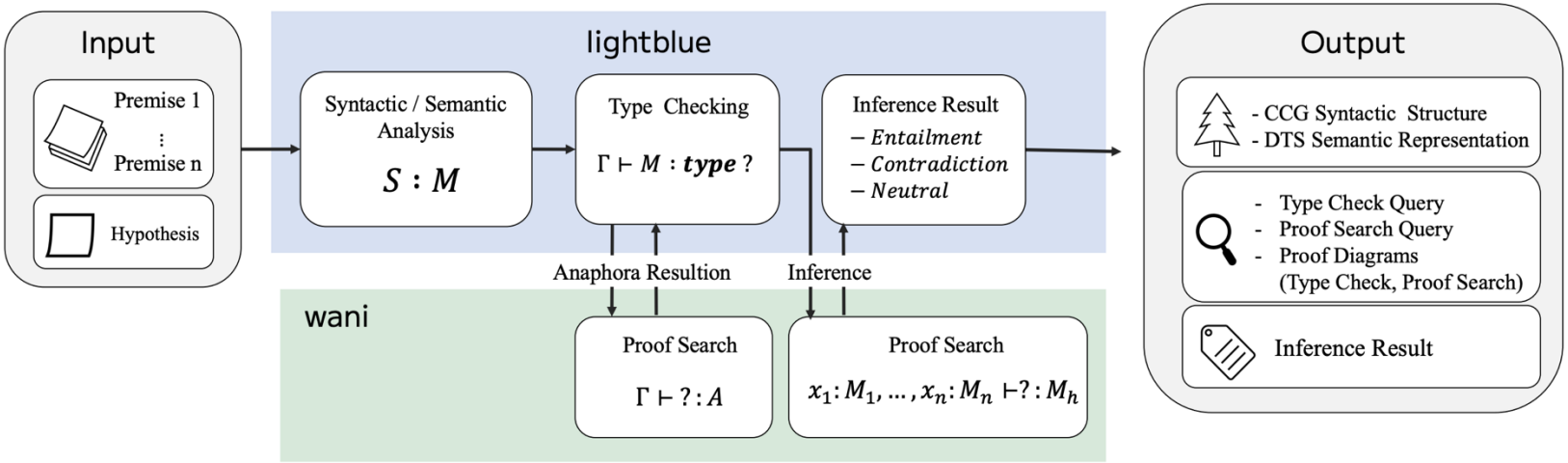
Koharu Saeki Daisuke Bekki
Ochanomizu University

BriGap-3 Université Paris Cité July 11, 2026

OUR PROJECT

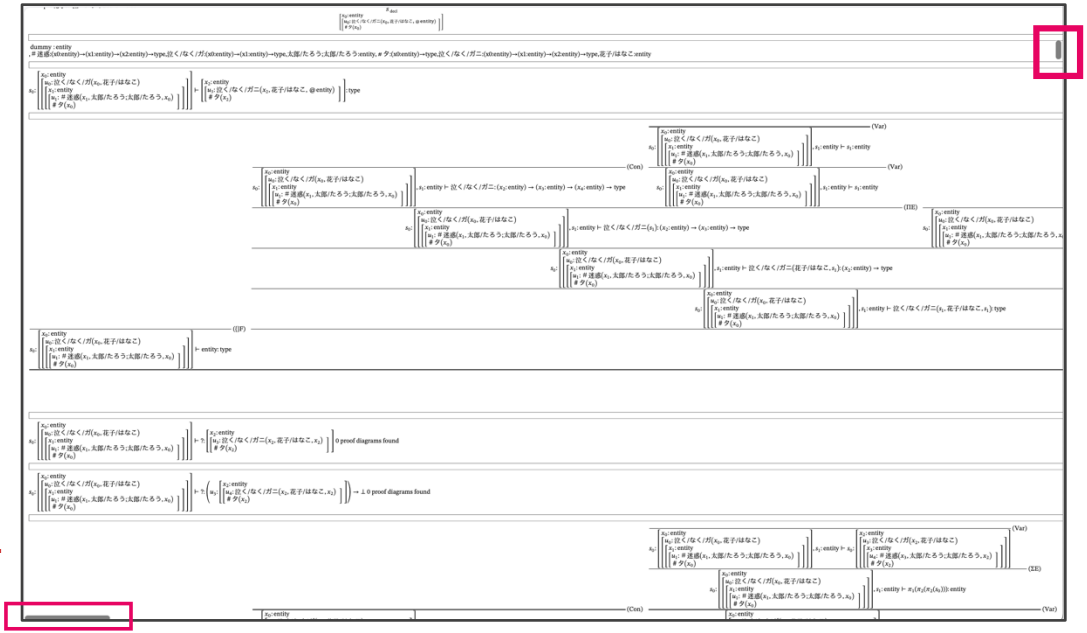
Our Natural Language Inference Pipeline

lightblue: a linguistically-oriented formal NLI system integrating **syntactic parsing**, **semantic composition**, **type checking**, and **proof search**.



Overview of the inference pipeline of *lightblue* (Tomita et al., 2025)

This is what a grammar developer sees today



Vertical scroll bar

Horizontal scroll bar

The grammar developer must find the single **path** they care about **by hand** in a 3.7 MB file.

one full route: syntactic/semantic parsing → type-checking → proof search

THE PROBLEM, TODAY (2/2)

Same problem, same engine — a different way of working

Today (exhaustive)		With Express	
260	**>120s**	**10**	**12s**
paths	backend time	paths	backend time

120s is the timeout cap: under exhaustive execution the search never finished.

We did not optimize the parser or prover.
We changed **who controls the reasoning process** — and **when computation happens**.

Why we choose formal NLI — and the catch

The appeal

Formal NLI grounds inference in **linguistic theory**, logical forms, and theorem proving, making inference **transparent** and **verifiable**, with explicit intermediate representations.

The catch

The ***rich linguistic representations*** that buy this transparency come back, **at development time**, as a **branching explosion**.

THE GAP (2/4)

Where the branches come from

lightblue (Tomita et al., 2025) … CCG (Steedman, 2000) + DTS (Bekki & Mineshima, 2017) — three sequential stages:

1. **Syntactic parsing and semantic composition:** CYK chart parsing → multiple admissible syntactic structures.
2. **Type checking & anaphora resolution:** underspecified terms trigger proof search → multiple outcomes per structure.
3. **Proof search** wani (Daido & Bekki, 2020) tests entailment / contradiction.

Key point

Type checking is **sequential**. Each outcome is the context for the next sentence ⇒ ambiguity **compounds across sentences**.

Running example: active → passive

Problem B

Premise
Taro-ga Hanako-o shikat-ta.
(Taro scolded Hanako.)

Hypothesis
Hanako-ga shikar-are-ta.
(Hanako was scolded.)

Expected label: **Yes**

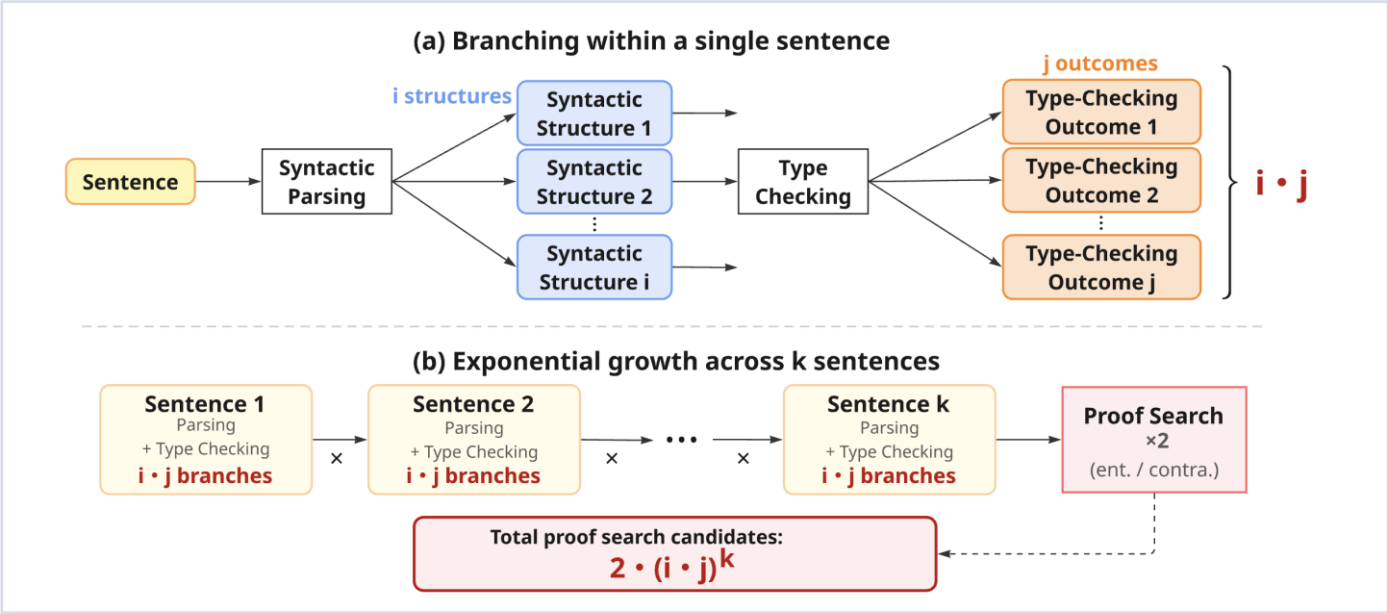
What has to be explored?

To verify the passive analysis, the developer must trace:
1. one syntactic structure
2. one type-checking outcome
3. the resulting proof-search query

Goal: trace **one theoretically interesting path**, not every possible analysis.

THE GAP (4/4)

The branching problem



$2 \cdot (i \cdot j)^k$ proof search queries.

Even **k=2, i=j=4** → **up to 512 queries** for a two-sentence problem.

The developer wants ***one***.

FROM EXECUTION TO EXPLORATION

The shift

Exhaustive Execution

compute everything,
then inspect

Selective Exploration

choose a branch,
then compute

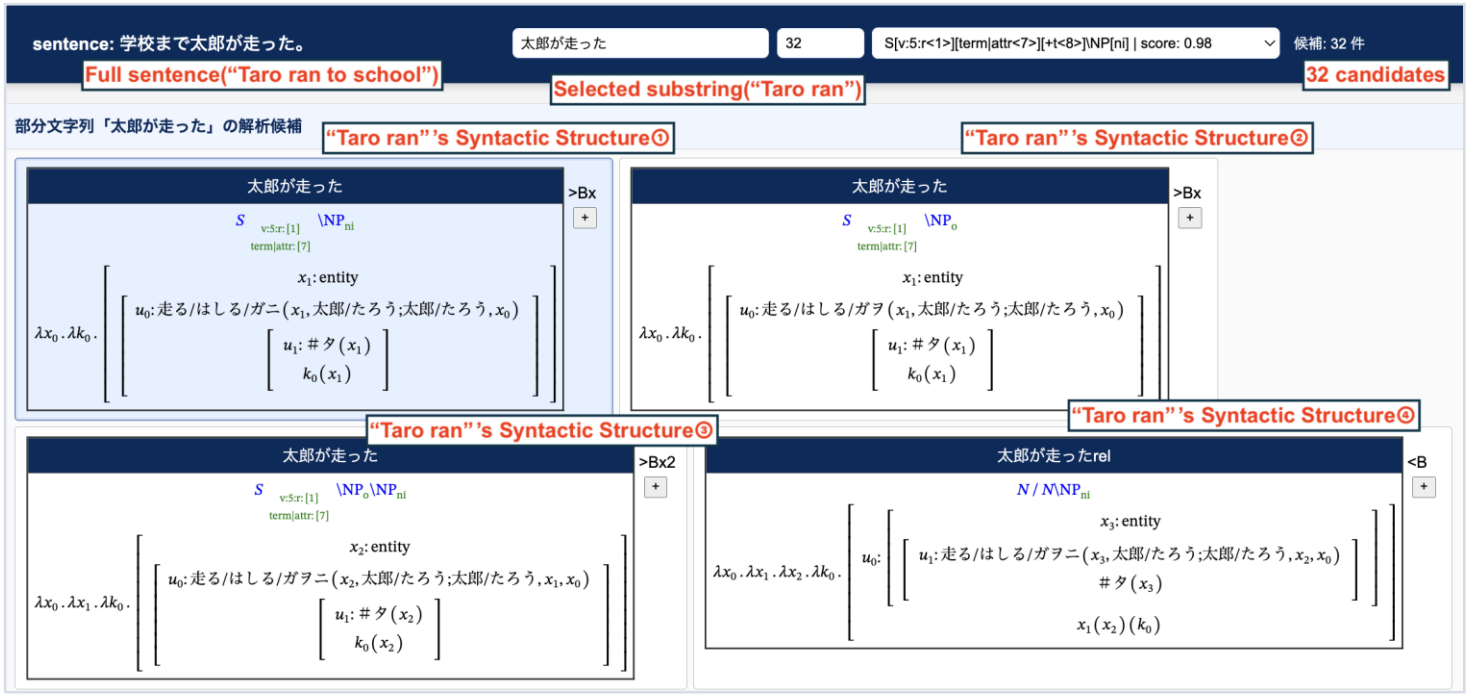
Under exhaustive execution, the developer's judgment never enters until computation is already over.

The developer selects ***one*** candidate at each branching point.

Active paths never grow to $2 \cdot (i \cdot j)^k$ — they stay **manageable at all times**.

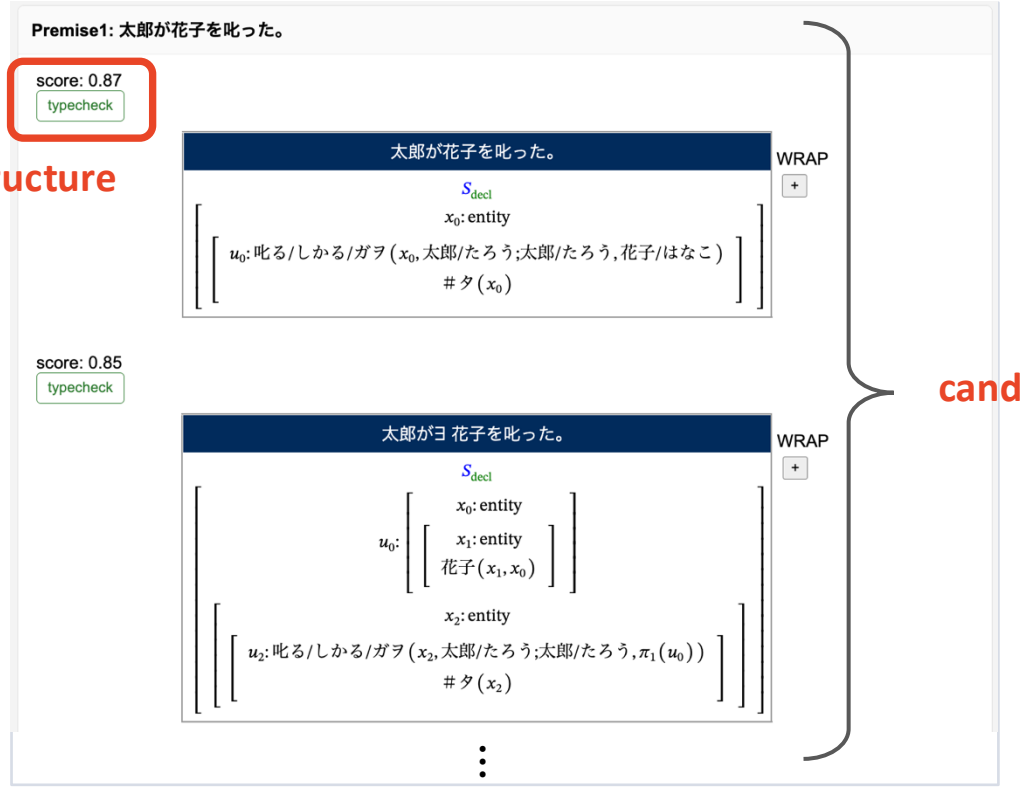
EXPRESS (1/3)

Investigate: localize to a substring



CYK retains structures for every span. Select ***“Taro ran”*** from *“Taro ran to school”* → candidates appear instantly, **without re-running the pipeline.**

Explore: choose, then compute(1/2)

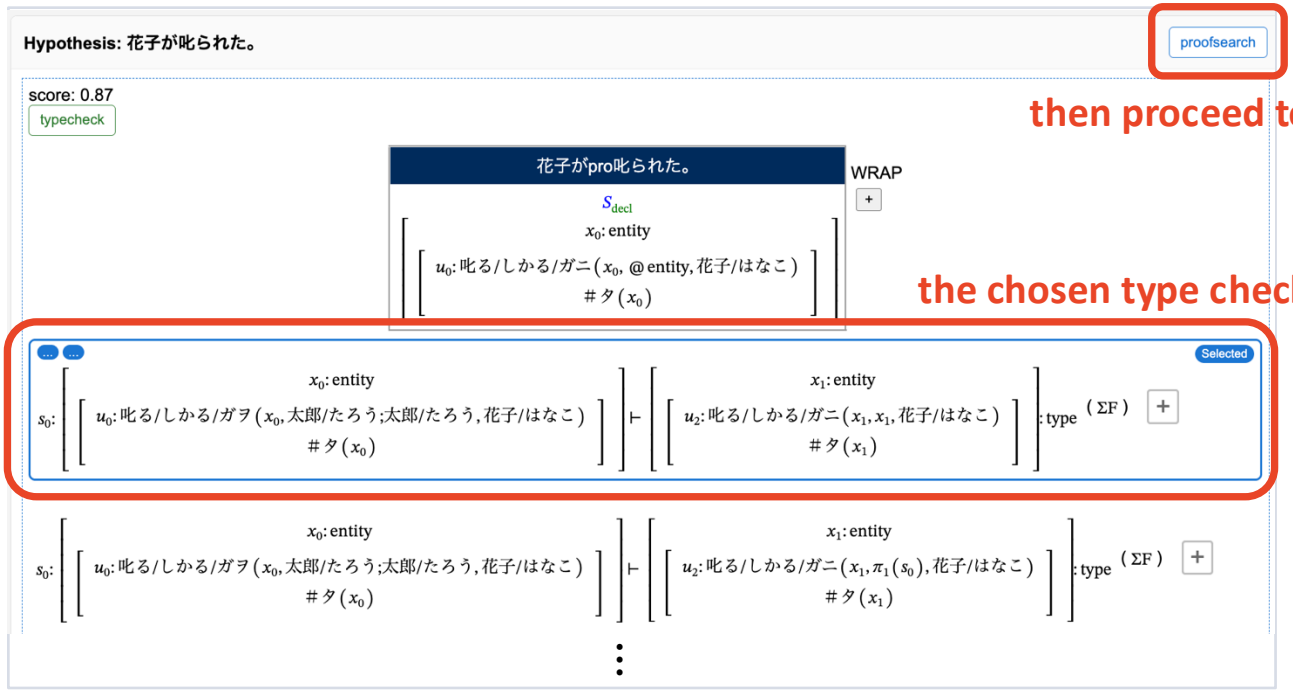


click to select this structure

candidates ranked by score

Pick one syntactic structure → proceed to type checking.

Explore: choose, then compute(2/2)

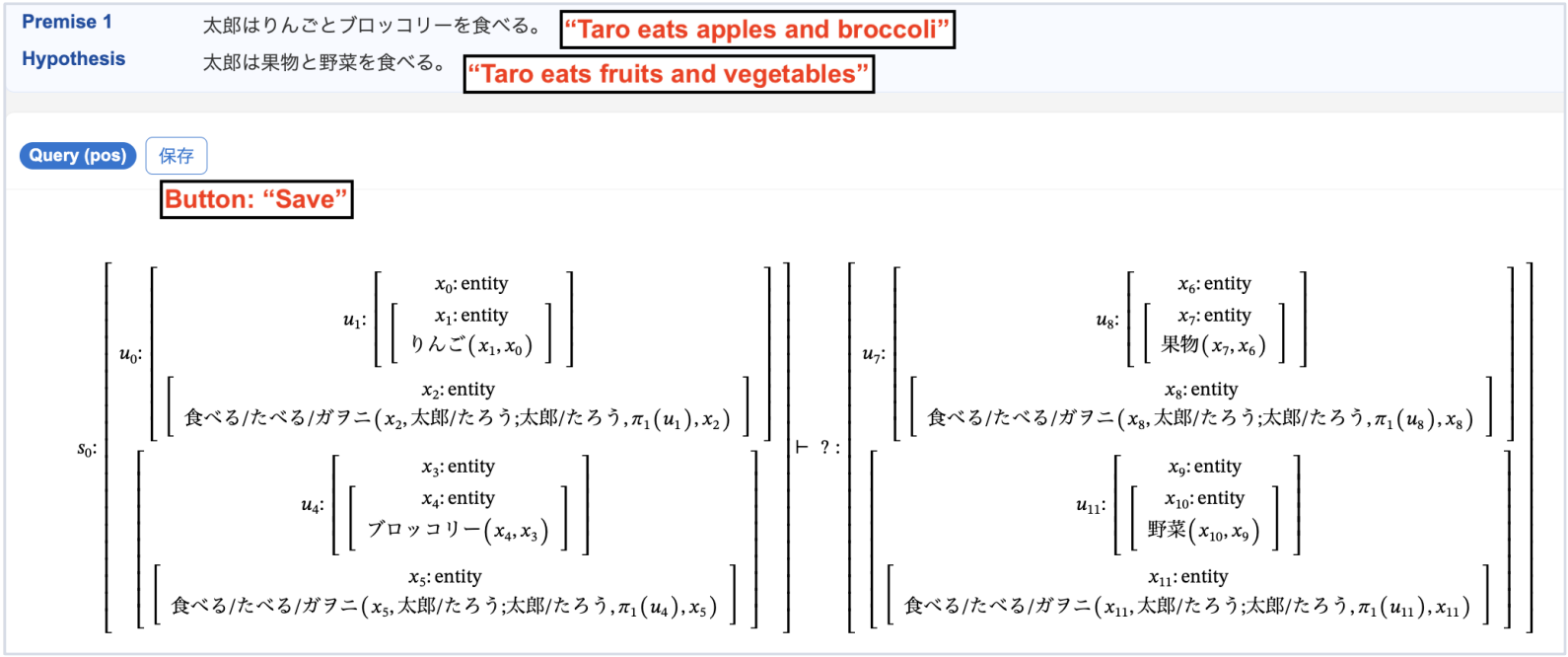


Pick one syntactic structure → one type-checking outcome → proof search.

One candidate per stage, per sentence ⇒ no exponential blow-up.

EXPRESS (3/3)

Share: export a reproducible query



The interactively formulated proof-search query is exported in a format **natively loadable by wani** — the prover developer reproduces the failure instantly.

DOES IT WORK? (1/2)

Search space: the number we promised

Two NLI problems, k=2, i=4, j=4 → up to **512 queries** under exhaustive execution.

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

Problem A: **96% fewer paths** (260 → 10).

Problem B: exhaustive never reached the target query; Express did, in 28 paths.

DOES IT WORK? (2/2)

Collaboration becomes possible

"Taro eats apples and broccoli" ⊨ "Taro eats fruits and vegetables"
→ **Unknown** (expected **Yes**).

1. **Grammar developer** narrows down to the problematic query in Express.
2. **Grammar developer** exports it in wani-loadable format and shares it.
3. **wani developer** reproduces the failure and pinpoints the root cause in minutes.

Result

Previously: manual transcription from screenshots (~1 hour). **≈ 12× faster debugging**, across developer roles.

CONCLUSION

The broader bridge

Formal NLI bridges **formal linguistics** and **computational linguistics**— theory made explicit and testable.

Why it matters

If the development workflow of the systems that embody this bridge stays impractical, **the bridge cannot be crossed**. By making that workflow tractable and transparent, ***Express*** keeps the connection practically sustainable.

Acknowledgments

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