NLIR: Natural Language Intermediate Representation for Mechanized Theorem Proving

Anonymous Author(s)
Affiliation
Address
email

Abstract

Formal theorem proving is challenging for humans as well as for machines. Thanks to recent advances in LLM capabilities, we believe natural language can serve as a universal interface for reasoning about formal proofs. In this paper, 1) we introduce *Pétanque*, a new lightweight environment to interact with the Coq theorem prover; 2) we present two interactive proof protocols leveraging natural language as an intermediate representation for designing proof steps; 3) we implement beam search over these interaction protocols, using natural language to rerank proof candidates; and 4) we use Pétanque to benchmark our search algorithms. Using our method with GPT-4o we can successfully synthesize proofs for 46% of the Logical Foundation series and for 50% of the first 100/260 lemmas from the newly published Busy Beaver proofs.[1]

1 Introduction

The general knowledge and reasoning abilities of frontier large language models (LLMs) makes them practical as a backbone for building agents able to interact with theorem provers. These agents should iteratively build proofs with help from proof engine feedback. While previous work (e.g. Yang et al. [2023]) used a costly data collection procedure to finetune modestly sized language models, we believe that reasoning in natural language before outputting tactics will lead to better and more interpretable results. Recently, Thakur et al. [2024] showed promising preliminary results by using GPT-4 as an agent proposing tactics inside a backtracking search and using rich feedback from the proof environment.

In this work, we develop infrastructure to allow communication between a GPT-4o-based agent and the Coq proof environment [The Coq Development Team, 2024]. Our key idea is to rely on natural language as much as possible when generating proofs. Using natural language leverages the strength of LLMs, and allows us to use chain-of-thought [Wei et al., 2022] by asking for an informal mathematical proof before generating the formal proof, making it more intuitive and comprehensible compared to purely automatic formal techniques. Additionally, partial proofs expressed in natural language are easier for humans to understand, adapt, or reuse, allowing for greater flexibility and collaboration between machine-generated suggestions and human mathematicians.

We present the following contributions: 1) *Pétanque*: A new fast and lightweight environment to interact with the Coq theorem prover. 2) Two interactive proof protocols both leveraging natural language reasoning: tactic-by-tactic proof construction, and hierarchical proof templating. 3) We couple both protocols with standard search algorithms leveraging feedback from the ITP and using natural language to rerank proof candidates. 4) We evaluate this agent on a new dataset of textbook

[1] `https://github.com/ccz181078/Coq-BB5`

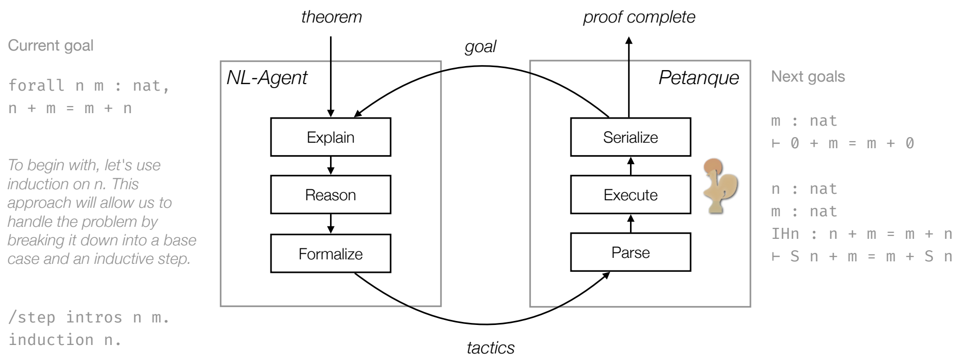

Figure 1: Tactic-by-tactic proof construction.

exercises and intermediate theorems from the recent Busy Beaver proof formalized in Coq of $\mathrm{BB}(4) = 107$, [ccz181078, 2024].

2 Pétanque: a lightweight interactive environment for Coq

A common difficulty when interacting with interactive proof assistants in the context of machine learning is inadequate tooling, see for example [Reichel et al., 2023]. Following existing work [Gallego Arias et al., 2016, Gallego Arias, 2019, Yang and Deng, 2019, Sanchez-Stern et al., 2020], we have built a new environment for machine to machine interaction for the Coq proof assistant, particularly tailored for interactive, high-throughput, low-latency learning applications. Pétanque is based on Flèche [Gallego Arias, 2024], a new document manager for Coq. We extend Flèche by enabling Pétanque to access the Coq proof engine directly without requiring edits in the associated document. This makes our environment fast and lightweight. A Python interface, `pytanque`, provides easy access to the API.

3 Proof interaction protocols

In this section, we present two approaches leveraging LLMs' ability to reason in natural language in order to find a formal proof with the help of a proof assistant. *Tactic-by-tactic proof construction* mimics the typical behavior of a standard Coq user: given the current goals, the agent generates one or several tactics that updates the goals and repeats this process until the proof is complete. By contrast, *hierarchical proof templating* tries to generate full proofs directly. Failed tactics are then replaced with *holes* to obtain a proof *template*. The agent then repeats the process of filling each hole until the proof is complete. Our approach's originality is that although both protocols' inputs (goals) and outputs (tactics) are in Coq code, the agent internally uses natural language as an intermediate representation to analyze the input and guide the code generation.

3.1 Tactic-by-tactic proof construction

An overview of the tactic-by-tactic proof construction agent is presented in Figure 1. Given a Coq theorem, the agent first uses natural language to describe the goal and explain how to continue the proof (chain-of-thought). The last step synthesizes the corresponding Coq tactics. For instance, in Figure 1, the goal is to prove that addition over natural numbers is commutative. The agent decides to try a proof by induction and correctly synthesizes a sequence of two tactics: `intros n m.` introduces two variables `n` and `m` of type `nat` (natural number), and `induction n.` starts an induction over `n`.

The tactics are sent to the Pétanque environment, which parses and executes each tactic to update the current goal. A textual representation of the new goal is then fed back to the agent, allowing it to progress further in the proof. If the execution returns an error, the current goal does not change, but we augment the prompt with the failed tactics and ask the LLM to try something else for the next attempt. For instance, in Figure 1, both tactics succeed and generate two new subgoals: the base case (for `n=0`, prove `m + 0 = 0 + m`) and the induction case (given the induction hypothesis

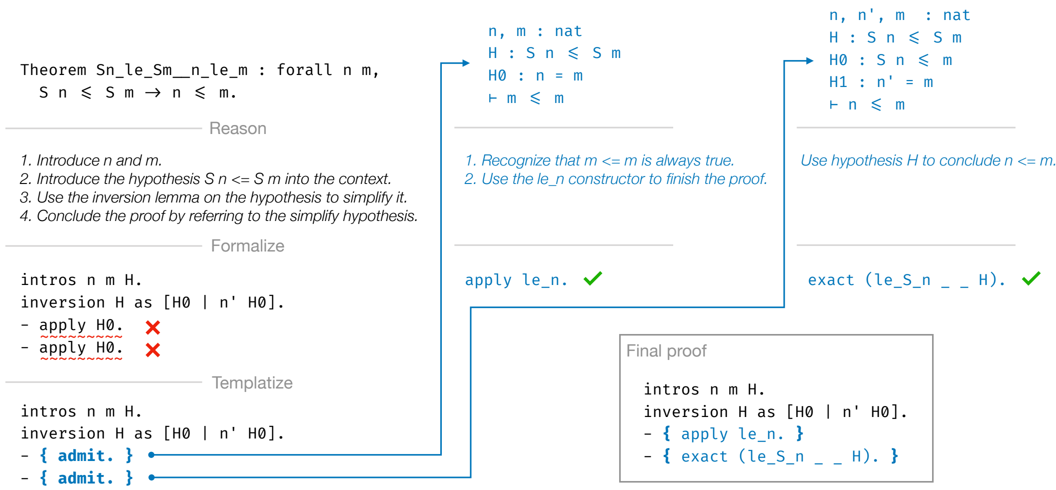

Figure 2: Hierarchical proof templating.

`IHn: n + m = m + n`, prove `(n + 1) + m = m + (n + 1)`. The textual representation of a goal uses the the symbol ⊢ to separate hypotheses from the conclusion, and `S n` denotes n + 1.

Model Interface. In early experiments, we observed that conversation-style reasoning often diverges: after a few rounds, the output makes very little sense, and the agent never recovers. Following Yang et al. [2024] – and similarly to Thakur et al. [2024] – we use a synthetic interface to summarize at each goal the global objective (initial theorem), the current goal (in the middle of a proof), and failed attempts to solve the same goal.

3.2 Hierarchical proof templating

An example execution of the hierarchical proof templating agent is presented in Figure 2. The agent pipeline is similar to the tactic-by-tactic method, but instead of focusing only on the next step, the agent generates a complete proof in natural language, before translating the proof in Coq syntax. For instance, in Figure 2, the agent uses the `inversion` tactics on the hypothesis `H` which generate two subgoals with a simpler hypothesis `H0`, and then tries to solve each subgoals using this `H0` hypothesis.

Then, rather than simply checking the proof, the Pétanque environment repairs it, by replacing failed tactics with *holes* which admits and closes the current subgoal, removing subsequent tactics until the focus moves to the next subgoal. Pétanque then checks that the resulting *template* is correct, i.e., assuming a valid proof for each holes, the proof is complete. A textual representation of each holes is then fed back to the agent which repeat the process to fill the holes one by one. For instance, in Figure 2, `apply H0` fails on both subgoals. The agent then repeats the process for each holes, using focused fine-grain reasoning to prove the corresponding subgoal. The proof is complete when there are no more holes.

4 Proof search

We combine our interactive protocol with the classic beam search algorithm. Inspired by Yao et al. [2023], we use the LLM to rank and sort the proposals at each step of the search.

A simplified version of the code is presented on the right. At each step, the `agent.generate` method generates multiple possible steps (tactics or proofs). Each step is then validated with the `petanque.step` method. and the state and the current proof of all the resulting candidates is stored. The `agent.sort` method then calls the LLM to discuss, compare and finally rank the candidates for the next step.

```python
def beam_search(n_steps, n_actions, beam_size):
  # Init
  s = petanque.start(thm)
  beam = [(s,[])] # (state, proofs) pairs
  for step in range(n_steps):
    # Generate candidates
    candidates = []
    for (s, p) in beam:
      # Try multiple actions for each state
      for a in agent.generate(s, n_actions)
        sa = petanque.step(s, a)
        pa = p + [a]
        # Proof found!
        if petanque.proof_finished(sa): return pa
        else: candidates = candidates + [(sa, pa)]
    # Rank candidates
    beam = agent.sort(candidates)[:beam_size]
  # No proof found
  return None
```

5 Evaluation

Logical Foundations exercises: We extracted the exercises of *Logical Foundations* [Pierce et al., 2024], the first volume of the *Software Foundation* textbooks series that is widely used to introduce Coq. We extracted 179 exercices. Given the popularity of this textbook the risk of data leak is high. We filtered out 66 "easy" exercises that are solved with one shot prompting (see **??** in **??**). This dataset thus comprises 113 exercises.

BB(4) lemmas: To avoid data leak issues, we extracted the 260 lemmas from the recent proof of $BB(4) = 107$ [ccz181078, 2024]. The repository was created in April 2024, long after the knowledge cutoff date of GPT-4o (October 2023). To provide the necessary context for the proof, for each lemma we augment the prompt with all the preceding definitions and lemmas.

Evaluation. The results are presented in the following table. The gray number in the *template* column indicates the number of proofs that were correct at the first try (no holes). On both dataset, we observe that the templating agent coupled with beam search

	Logical Foundations				$BB(4)$	
	tactics		template		template	
	naive	beam	naive	beam	naive	beam
% success	30.1	40.7	(16.8) 29.2	(23.0) **46.0**	(24.0) 35.0	(40.0) **50.0**

We use Coq 8.19.2 and GPT-4o (Sept. 2024) for all experiments. We observe that the template agent coupled with beam search (`n_steps=10, n_actions=4, beam_size=3`) outperforms the tactic agent on the Logical Foundation benchmark. To limit the costs of our experiments, we only run the template agent on the first 100 Lemmas of the $BB(4)$ benchmark. For the template agent, the gray numbers indicate the proportion of proofs that are correct at the first try (no holes).

6 Related work and conclusion

LLMs and theorem provers Automatic theorem-proving is a longstanding challenge in computer science Newell et al. [1957]. Recent work has used neural models based on autoregressive language model that generate a proof tactic by tactic. Most works use finetuned LLMs [Polu and Sutskever, 2020, Han et al., 2021, Wu et al., 2022, Yang et al., 2023, First et al., 2023], trained on (goal, tactic) pairs obtained from intermediate steps of existing proofs. On the other hand, Lample et al. [2022] uses online training, progressively collecting more data. Closest to our work, Thakur et al. [2024] build a tactic-by-tactic LLM agent based on GPT-4 and also use an interface to summarize past interactions. They, however, do not use proof repair or beam search. Other work close to ours is Wang et al. [2024], who use proof repair over hierarchical proofs in Isabelle, coupled with best-first search. Contrary to us, they use fine-tuned models and no chain-of-thought.

Reasoning in LLMs This work is also related to recent investigations on the reasoning abilities of LLMs [Plaat et al., 2024]. Chain-of-Thought (CoT) prompting [Wei et al., 2022] was shown to improve LLM's answers; subsequent work found that these reasoning abilities could be elicited zero-shot [Kojima et al., 2022]. Further work interleaved CoT with decision-making [Yao et al., 2022], added search and complex control flow to reasoning [Chen et al., 2022, Yao et al., 2023, Besta et al., 2024], incorporated refinement and feedback [Madaan et al., 2024, Shinn et al., 2024], and learned to generate novel reasoning traces that proved beneficial for further training [Zelikman et al., 2022, 2024]. Like our work, many of these methods – especially the ones using search and refinement – make use of LLM-based scoring or ranking functions [Zheng et al., 2023].

Conclusion In this work, we have presented a new agent for building proofs leveraging chain of thought as an intermediate representation, and generating proofs by outputting step-by-step tactics or hierarchical proof templates. We couple this with beam search and natural language reranking and obtain good performance on a new evaluation set built with the help of our novel proof environment, *Pétanque*. Future work could investigate how one could use reinforcement learning to obtain better reasoning and performance with smaller models [OpenAI, 2024].

References

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2024.

ccz181078. https://github.com/ccz181078/Coq-BB5/tree/main, 2024.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. *CoRR*, abs/2303.04910, 2023.

Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jscoq: Towards hybrid theorem proving interfaces. In Serge Autexier and Pedro Quaresma, editors, *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016*, volume 239 of *EPTCS*, pages 15–27, 2016. doi: 10.4204/EPTCS.239.2. URL https://doi.org/10.4204/EPTCS.239.2.

Emilio Jesús Gallego Arias. SerAPI: Machine-friendly, data-centric serialization for Coq. preprint, 01 2019. URL https://github.com/ejgallego/coq-serapi/.

Emilio Jesús Gallego Arias. Flèche: Incremental validation for hybrid formal documents. under revision, 2024.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*, 2021.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

Allen Newell, John Clifford Shaw, and Herbert A Simon. Empirical explorations of the logic theory machine: a case study in heuristic. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 218–230, 1957.

OpenAI. Learning to Reason with LLMs. https://openai.com/o1/, 2024.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2024. Version 6.7, http://softwarefoundations.cis.upenn.edu.

Aske Plaat, Annie Wong, Suzan Verberne, Joost Broekens, Niki van Stein, and Thomas Back. Reasoning with large language models, a survey. *arXiv preprint arXiv:2407.11511*, 2024.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020.

Tom Reichel, R. Wesley Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer. Proof repair infrastructure for supervised models: Building a large proof repair dataset. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 26:1–26:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPICS.ITP.2023.26. URL https://doi.org/10.4230/LIPIcs.ITP.2023.26.

Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *MAPL@PLDI*, 2020.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Amitayush Thakur, George D. Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. In *COLM*, 2024.

The Coq Development Team. The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman, 2024.

Haiming Wang, Huajian Xin, Zhengying Liu, Wenda Li, Yinya Huang, Jianqiao Lu, Zhicheng Yang, Jing Tang, Jian Yin, Zhenguo Li, and Xiaodan Liang. Proving theorems recursively. *CoRR*, abs/2405.14414, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *NeurIPS*, 2022.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *CoRR*, abs/2405.15793, 2024.

Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *ICML*, 2019.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. In *NeurIPS*, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *NeurIPS*, 2023.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *NeurIPS*, 2023.

A Prompts

A.1 Tactic-by-tactic proof construction prompt example

Instructions

You are an analytical and helpful assistant proficient in mathematics as well as in the use of the Coq theorem prover and programming language. You will be provided with a Coq/math-comp theorem and your task is to prove it. This will happen in interaction with a Coq proof engine which will execute the proof steps you give it, one at a time, and provide feedback. This is the important information about this task:

Coq engine interface

You will be provided with:

- This information prompt;
- The theorem to prove;
- Successful proof steps until now (current proof);
- Unsucessful proof step attempts with the current goal(s), if any; you know these techniques didn't work, so try avoid reusing them;
- The current goal;

Interaction

Your goal is to write proof steps interactively until you manage to find a complete proof for the proposed theorem. You will be able to interact with the proof engine by issuing the following commands:

Step : Passes the string that is given after it to the Coq proof engine. Example usage:

```
/step intros.
```

You can use several steps in each interaction, but try to be concise and advance one step at a time, especially if you've been getting errors.

Theorem and proof information

You have interacted 2 times with the engine.

Theorem

Here is the theorem to prove:

```
forall f : nat -> nat,
(forall n : nat, n = f (f n)) -> forall n1 n2 : nat, f n1 = f n2 -> n1 = n2
```

Proof

Here are the proof steps until now:

```
intros f H n1 n2 H0.
```

Previous unsuccessful steps

Here are the previous unsuccessful proof step attempts. These have all been tried before with the current goal(s). DOT NOT TRY ANY OF THESE STEPS, as you know they don't work. You should try something different.

```
rewrite H0.
```

Current goal(s)

```
f : nat -> nat
H : forall n : nat, n = f (f n)
n1 : nat
n2 : nat
|- a2 f n1 = f n2 -> n1 = n2
```

A.2 Hierarchical proof templating prompt example

Your task is to complete a proof using the Coq proof assistant. For each theorem, I will give you the goal to prove in Coq syntax.

Here are a few examples:

```
<example>
<goal>
n, m, p : nat
|- nat, n + (m + p) = m + (n + p)
</goal>

<proof>
rewrite Nat.add_assoc. rewrite Nat.add_assoc.
assert (n + m = m + n) as H by apply Nat.add_comm.
rewrite H. reflexivity.
</proof>
</example>
```

[...]

Think before you write the proof in <thinking> tags. First explain the goal. Then describe the proof step by step. Finally write the corresponding Coq proof in <proof> tags using your analysis. Do not repeat the context and do no restate the theorem.

You are in the middle of the proof of `involution_injective`:

```
forall f : nat -> nat,
(forall n : nat, n = f (f n)) -> forall n1 n2 : nat, f n1 = f n2 -> n1 = n2
```

Ready? Here is the current goal.

```
<goal>
f : nat -> nat
H : forall n : nat, n = f (f n)
n1 : nat
n2 : nat
Hf_eq : f n1 = f n2
|- n1 = n2
</goal>
```

Take a deep breath and walk me through the proof.

B Detailed results

B.1 Logical Foundations

For the template agent, the gray numbers indicate the proportion of proofs that are correct at the first try (no holes). We also report the average length of the generated proof (number of tactics) and the size of the smallest and the biggest proof.

	tactics		template	
	naive	beam	naive	beam
andb_true_elim2	6	5	10	10
lower_letter_lowers	x	8	27	8
grade_lowered_once	x	8	9	15
eqblist_refl	x	x	x	x
count_member_nonzero	x	x	x	x
remove_does_not_increase_count	x	x	x	x
involution_injective	7	x	9	7
option_elim_hd	x	x	x	x
eqb_id_refl	x	6	22	18
update_eq	14	12	x	14
update_neq	7	7	7	7
add_comm	x	12	x	x
even_S	x	x	x	41
add_shuffle3	x	x	x	x
mul_comm	x	x	x	x
plus_leb_compat_l	x	x	x	x
mult_plus_distr_r	x	x	x	x
mult_assoc	x	12	x	x
add_shuffle3'	13	x	x	x
bin_to_nat_pres_incr	x	x	x	x
nat_bin_nat	x	x	x	x
bin_nat_bin	x	x	x	x
optimize_0plus_b_sound	x	x	x	x
pup_to_2_ceval	x	x	x	x
loop_never_stops	x	x	x	x
no_whiles_eqv	x	x	x	x
execute_app	x	x	x	x
s_compile_correct	x	x	x	x
break_ignore	4	4	4	4
while_continue	4	4	6	6
while_stops_on_break	x	4	x	x
seq_continue	x	x	x	x
seq_stops_on_break	4	4	x	x
while_break_true	4	4	x	6
ceval_deterministic	x	8	9	3
ev_double	8	8	13	11
ev5_nonsense	6	7	x	21
ev'_ev	x	x	x	x
ev_plus_plus	x	x	x	x
total_relation_is_total	x	x	x	x
empty_relation_is_empty	5	5	x	5
O_le_n	4	4	9	4
Sn_le_Sm__n_le_m	5	8	5	10
lt_ge_cases	x	7	x	x
le_plus_l	6	6	x	10
plus_le	x	x	x	x
add_le_cases	x	14	x	x
plus_le_compat_r	x	14	x	9
le_plus_trans	6	6	x	10
n_lt_m__n_le_m	x	7	11	8
plus_lt	x	x	x	x
leb_complete	x	x	23	25
leb_correct	x	x	x	x
leb_true_trans	7	7	x	9
R_equiv_fR	x	x	x	x
subseq_refl	x	x	x	x
subseq_app	4	4	4	4
subseq_trans	x	4	6	8
reflect_iff	13	12	18	16
eqbP_practice	x	18	x	x
merge_filter	x	4	4	6
pal_app_rev	x	x	x	x
pal_rev	7	4	4	4
palindrome_converse	x	x	x	x
pigeonhole_principle	x	x	x	x
regex_match_correct	x	x	x	x
rev_involutive	10	13	12	12
map_rev	x	x	x	x
uncurry_curry	x	x	x	x
curry_uncurry	x	x	x	x
ceval__ceval_step	x	x	x	x
leb_plus_exists	x	x	x	x
In_map_iff	31	x	x	38
In_app_iff	x	x	52	55
All_In	x	x	x	x
combine_odd_even_intro	x	x	x	x
combine_odd_even_elim_odd	x	x	x	x
combine_odd_even_elim_even	x	x	x	x
eqb_neq	x	x	x	x
eqb_list_true_iff	x	x	x	x
forallb_true_iff	x	x	x	x
tr_rev_correct	x	x	x	x
excluded_middle_irrefutable	18	4	9	9
total_relation_not_partial_function	x	x	x	x
lt_trans'	15	8	x	x
lt_trans''	11	12	x	18
le_S_n	x	5	15	6
le_not_symmetric	10	x	x	13
le_antisymmetric	14	8	x	11
le_step	9	x	x	11
rtc_rsc_coincide	x	x	x	32
booltree_ind_type_correct	x	x	x	x
Toy_correct	x	7	x	x
reflect_involution	x	x	x	x
t_update_neq	7	x	24	11
t_update_permute	x	x	x	x
rev_exercise1	9	6	8	6
eqb_true	x	x	29	24
plus_n_n_injective	x	x	x	37
combine_split	x	x	29	18
bool_fn_applied_thrice	x	x	33	69
eqb_sym	x	x	21	19
eqb_trans	x	x	x	x
split_combine	x	x	x	x
existsb_existsb'	x	x	x	x
ev_8	7	7	7	7
pe_implies_pi	x	x	x	13
ev100	x	21	4	5
andb3_exchange	x	20	x	40
andb_true_elim2	4	3	8	8
andb3_exchange'	x	5	x	14
nor_comm'	14	12	x	19
nor_not'	11	9	19	x

Table 1: Detailed results for the Logical Foundations benchmark.

	tactics		template		
	naive	beam	naive	beam	total
# success	34	46	(19) 33	(26) **52**	113
% success	30.1	40.7	29.2	**46.0**	100.0
average proof length	9.1	8.13	14.4	15.4	
(min, max) proof length	(4, 31)	(4, 21)	(4, 52)	(3, 69)	

B.2 $BB(4)$

For each methods, we also report the original proof sizes (mean, min, and max) on the set of lemmas that was successfully proved.

	orig.	naive	beam
`ffx_eq_x_inj`	10	9	7
`enc_v1_eq`	6	x	x
`enc_pair_inj`	12	x	x
`enc_list_inj`	16	x	x
`andb_shortcut_spec`	3	7	7
`orb_shortcut_spec`	3	9	7
`set_ins_spec`	33	x	x
`empty_set_WF`	10	24	18
`pop_back_len`	8	x	x
`pop_back__nth_error`	15	x	x
`list_eq__nth_error`	34	48	x
`pop_back'__push_back`	6	x	x
`St_enc_inj`	2	37	37
`St_eqb_spec`	3	3	3
`Sigma_eqb_spec`	3	x	x
`Sigma_enc_inj`	2	x	x
`listSigma_inj`	12	x	48
`map_inj`	9	27	26
`listT_enc_inj`	7	7	8
`Dir_eqb_spec`	3	3	13
`St_list_spec`	4	12	26
`Sigma_list_spec`	4	13	13
`Dir_list_spec`	4	x	13
`forallb_St_spec`	9	x	14
`forallb_Sigma_spec`	9	x	33
`forallb_Dir_spec`	9	17	16
`Steps_trans`	9	x	18
`Steps_unique`	11	22	x
`Steps_NonHalt`	22	x	x
`HaltsAt_unique`	16	x	x
`NonHalt_iff`	27	x	x
`LE_step`	10	39	18
`LE_Steps`	10	34	x
`LE_NonHalts`	8	x	x
`HaltTimeUpperBound_LE_NonHalt`	7	x	x
`LE_HaltsAtES_1`	11	x	x
`LE_HaltsAtES_2`	14	x	x
`HaltTimeUpperBound_LE_Halt`	15	x	x
`St_swap_swap`	12	x	x
`Trans_swap_swap`	7	x	8
`option_Trans_swap_swap`	7	10	11
`TM_swap_swap`	8	x	9
`ExecState_swap_swap`	7	6	6
`step_swap`	18	x	x
`step_halt_swap`	10	27	x
`Steps_swap`	27	x	x
`LE_swap_0`	7	x	16
`LE_swap`	9	x	x
`InitES_swap`	8	x	12
`HaltsAt_swap_0`	15	19	x
`HaltsAt_swap`	9	x	x
`HaltTimeUpperBound_LE_swap`	10	x	x
`HaltTimeUpperBound_LE_swap_InitES`	5	x	x
`Trans_rev_rev`	7	15	7
`option_Trans_rev_rev`	8	10	11
`TM_rev_rev`	7	10	9
`Tape_rev_rev`	7	x	10
`ExecState_rev_rev`	7	x	7
`fext_inv`	3	5	5
`step_rev`	44	x	x
`step_halt_rev`	11	x	x
`Steps_rev`	27	x	x
`LE_rev_0`	7	x	16
`LE_rev`	9	x	x
`InitES_rev`	3	13	5
`HaltsAt_rev_0`	15	17	17
`HaltsAt_rev`	9	x	31
`HaltTimeUpperBound_LE_rev`	10	x	x
`HaltTimeUpperBound_LE_rev_InitES`	5	x	x
`Trans_swap_id`	10	x	x
`isUnusedState_spec`	58	x	x
`step_UnusedState`	11	14	15
`Steps_UnusedState`	15	x	x
`HaltTimeUpperBound_LE_HaltsAtES_UnusedState`	68	x	x
`TM0_LE`	7	5	5
`UnusedState_TM0`	10	22	21
`UnusedState_dec`	4	x	10
`HaltTimeUpperBound_LE_HaltAtES_MergeUnusedState`	31	x	x
`St_to_nat_inj`	4	4	5
`St_suc_le`	4	x	23
`St_suc_eq`	5	x	13
`St_suc_neq`	3	7	10
`HaltTimeUpperBound_LE_HaltAtES_UnusedState_ptr`	21	x	x
`HaltsAtES_Trans`	27	x	17
`UnusedState_upd`	68	x	x
`UnusedState_ptr_upd`	97	x	x
`isHaltTrans_0`	3	17	20
`CountHaltTrans_upd`	7	x	x
`CountHaltTrans_0_NonHalt`	21	x	x
`Trans_list_spec`	6	x	x
`St_leb_spec`	13	x	13
`TM_simplify_spec`	6	7	16
`TM_upd'_spec`	5	9	5
`nat_eqb_spec`	3	x	11
`TNF_Node_expand_spec`	64	x	x
`TNF_Node_NonHalt`	6	x	15
`HaltDecider_cons_spec`	7	x	18
`SearchQueue_upd_spec`	74	x	x
`SearchQueue_upd_bfs_spec`	30	x	x
`SearchQueue_reset_spec`	13	38	x

Table 2: Detailed results for the $BB(4)$ benchmark.

	template		
	naive	beam	total
# success	(19) 35	(40) **50**	113
% success	35.0	**40.0**	100.0
average proof length	16.2	14.4	
original average proof length	7.9	7.1	
(min, max) proof length	(3, 48)	(3, 48)	
original (min, max) proof length	(2, 34)	(2, 27)	