A Language-Agent Approach to Formal Theorem-Proving

Amitayush Thakur, Yeming Wen & Swarat Chaudhuri
Department of Computer Science
The University of Texas at Austin
Austin, TX, USA
`{amitayush, ywen}@utexas.edu, swarat@cs.utexas.edu`

Abstract

Language agents, which use a large language model (LLM) capable of in-context learning to interact with an external environment, have emerged as a promising approach to control tasks. We present a language-agent approach that offers state-of-the-art performance in formal theorem-proving. Our method, COPRA, uses a high-capacity, black-box LLM (GPT-4) as part of a policy for a stateful backtracking search. During the search, the policy can select proof tactics and retrieve lemmas and definitions from an external database. Each selected tactic is executed in the underlying proof framework, and the execution feedback is used to build the prompt for the next policy invocation. The search also tracks selected information from its history and uses it to reduce hallucinations and unnecessary LLM queries. We evaluate COPRA on the `miniF2F` benchmark for Lean and a set of Coq tasks from the Compcert project. On these benchmarks, COPRA is significantly better than one-shot invocations of GPT-4, as well as state-of-the-art models fine-tuned on proof data, at finding correct proofs quickly.

1 Introduction

Automatically proving formal theorems (Newell et al., 1957) is a longstanding challenge in computer science. Autoregressive language models (Polu & Sutskever, 2020; Han et al., 2021; Yang et al., 2023) have recently emerged as an effective approach to this problem.

A weakness of this method is that it does not model the *interaction* between the model and the underlying proof framework. The application of a tactic is an *action* that changes the state of the proof and the interpretation of future tactics. By ignoring these game-like dynamics, autoregressive models miss out on a valuable source of feedback and end up being more susceptible to hallucinations.

In this paper, we show that the nascent paradigm of *large-language-model (LLM) agents* (Yao et al., 2022; Wang et al., 2023; Shinn et al., 2023) can help address this weakness. Here, one uses an LLM as a *agent* that interacts with an external environment. Information gathered through interaction is used to update the LLM's prompt, eliciting new agent behavior because of in-context learning.

Our approach, called COPRA[1] (Figure 1), uses an off-the-shelf, high-capacity LLM (GPT-4 (OpenAI, 2023)) as part of a policy in that interacts with a proof environment like Coq or Lean. At each time step, the policy consumes a textual prompt and chooses to use an available tactic, or backtrack, or retrieve relevant lemmas and definitions from an external corpus. When the policy selects a tactic, we "execute" it using the underlying proof assistant. The feedback from the execution is used to construct a new prompt for the policy, and the process repeats.

[1]COPRA is an acronym for "In-**co**ntext **Pr**over **A**gent".

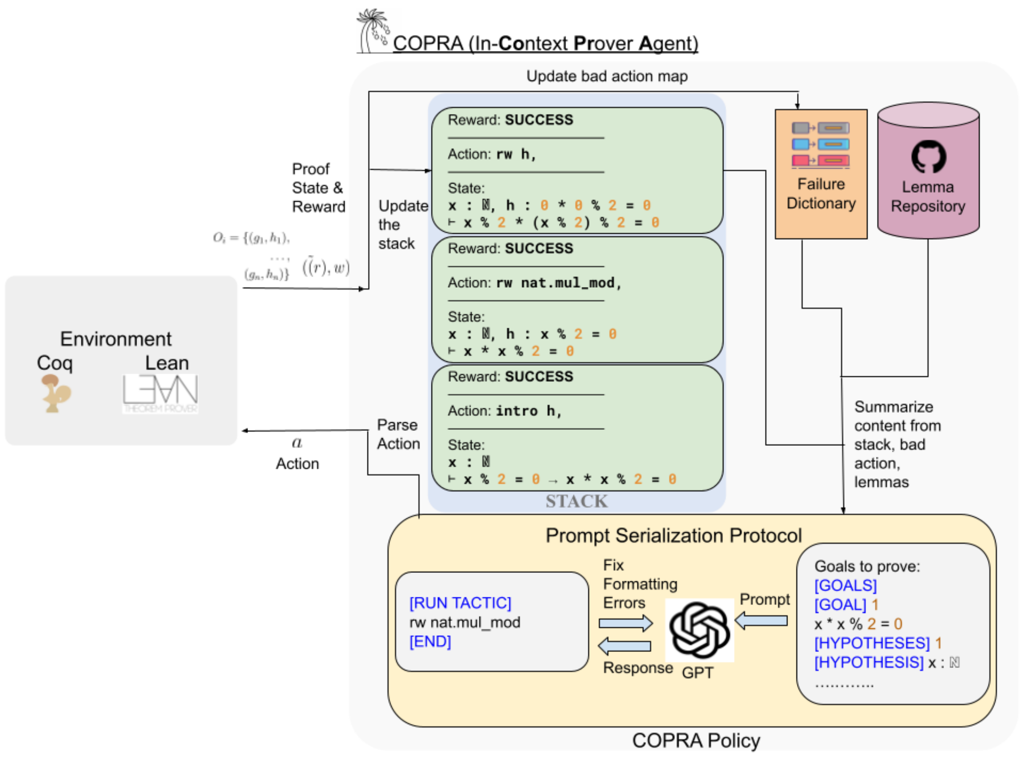


Figure 1: An overview of COPRA. The system implements a policy that interacts with a proof environment (Coq or Lean). Internally, a COPRA policy consists of an LLM (GPT-4), a stack-based backtracking search, a retrieval mechanism, a dictionary tracking past failures, and a prompt serialization protocol that constructs LLM prompts using the stack and environment feedback and parse LLM outputs into actions.

We have integrated COPRA with both the Coq and the Lean environments. We evaluate the system using the `miniF2F` Zheng et al. (2021) benchmark for competition-level mathematical reasoning in Lean and a set of Coq proof tasks (Sanchez-Stern et al., 2020) from the Compcert (Leroy, 2009) project on verified compilation. Using a new metric called *prove-at-k-inferences*, we show that COPRA can converge to correct proofs faster than competing approaches, including the state-of-the-art models (Yang et al., 2023; Sanchez-Stern et al., 2020) trained on formal proof data. We also show that when COPRA fails, it fails quicker than the baseline methods.

1.1 Problem Formulation

COPRA is based on a view of automatic theorem-proving as a *control problem*. Like prior work on reinforcement learning (RL) for proof synthesis (Wu et al., 2021), we view a theorem-prover as a *policy* that interacts with a stateful proof environment (e.g., Lean) and model the interaction between the policy and the environment as a deterministic Markov Decision Process (MDP). We depart from prior RL-based work for theorem-proving by imposing a partial order on MDP states, allowing rewards to have a textual component, and allowing history-dependent policies.

Now we describe the different components of our *proof MDP*

States. Let an *obligation* be a pair (g, h), where g is a goal and h a hypothesis. A *state* of the MDP is either a special symbol called *error* or a set $O = \{o_1, \ldots, o_k\}$ of obligations o_i. The MDP has a unique *initial state* o_{in} with a single obligation (g_{in}, h_{in}), where the goal g_{in} and the hypothesis h_{in} are extracted from the user-provided theorem that we are trying to prove. Its unique *final state* `QED` is the empty obligation set.

Following Sanchez-Stern et al. (2020), we define a partial order $\sqsubseteq$ over states that defines when a state is "at least as hard" than another and use it to avoid actions that do not lead to progress in the proof. Formally, for states O_1 and O_2 with $O_1 \neq error$ and $O_2 \neq error$, $O_1 \sqsubseteq O_2$ iff

$$\forall\, o_i = (g_i, h_i) \in O_1.\ \exists o_k = (g_k, h_k) \in O_2.\ g_k = g_i \wedge (h_k \rightarrow h_i).$$

Intuitively, $O_1 \sqsubseteq O_2$ if for every obligation in O_1, there is a stronger obligation in O_2. We assume we have an efficient symbolic procedure that can check this relationship for any pair of states. The procedure is *sound*, meaning that if it reports $O_1 \sqsubseteq O_2$, the relationship actually holds. However, it is *incomplete*, i.e., it may not detect all relationships of the form $O_1 \sqsubseteq O_2$.

Actions and Transitions. The actions in our MDP are the proof environment's *tactics*. The transition function $T(O, a)$ determines the result of applying an action a to a state O. When a is a tactic, we assume the underlying proof environment to return a state O' that results from applying a to O. If a is a "bad" tactic, then O' equals $error$; otherwise, O' is a new set of obligations. We assume that our agent can evaluate $T(O, a)$ for any state O and action a. While this assumption is unacceptable in many MDP problems, it is reasonable in the theorem-proving setting.

Rewards. As usual, we assume a *reward function* $R(O, a)$ that evaluates an action a at a state O. Historically, such functions are scalar-valued; however, because we use LLMs as policies, we allow rewards to also include rich textual feedback from the proof environment. Concretely, we consider rewards of the form $R(O, a) = (\tilde{r}, w)$, where: (1) $\tilde{r}$ is a very high positive value if $T(O, a) = \texttt{QED}$, a negative value if $T(O, a) = error$, and 0 otherwise, and (2) w is the feedback from the proof environment when a is executed from O.

2 The COPRA Agent

A COPRA policy has access to an LLM (in practice, GPT-4) and performs a depth-first search. During the search, it records information about failed actions. It also uses the $\sqsubseteq$ relation over states to checks that it is making progress on the proof.

```
COPRA($O$)
1   PUSH($st, O$)
2   $\rho \leftarrow$ RETRIEVE($O$)
3   for $j \leftarrow 1$ to $k$
4       do $p \leftarrow$ PROMPTIFY($st, Bad(O), \rho, r$)
5           $a \sim$ PARSEACTION(LLM($p$))
6           $O' \leftarrow T(O, a)$, $r \leftarrow R(O, a)$
7           if $O' = $ QED
8              then terminate successfully
9              else if $O' = error$ or
                        $\exists O'' \in st.\ O'' \sqsubseteq O'$
10                     then add $a$ to $Bad(O)$
11                     else COPRA($O'$)
12  POP($st$)
```

Figure 2: The search procedure in COPRA. T is the environment's transition function and R is the reward function. st is a stack, initialized to be empty. $Bad(O)$ is a set of actions, initialized to $\emptyset$, that are known to be bad at O. LLM is an LLM, PROMPTIFY generates a prompt, PARSEACTION parses the output of the LLM into an action (repeatedly querying the LLM in case there are formatting errors in its output), and RETRIEVE gathers relevant lemmas and definitions from an external source. The procedure is initially called with argument O_{in}.

Figure 2 shows pseudocode for such a policy. The policy maintains a stack of MDP states and a "failure dictionary" Bad that maps a state to a set of actions that are known to be "unproductive" at the state. At each search step, the algorithm pushes the current state on the stack and retrieves external lemmas and definitions relevant to the state. After this, it repeatedly serializes the stack and $Bad(O)$ into a prompt and feeds it to the LLM. The LLM's output is parsed into an action, and the agent executes it in the environment.

One outcome of the action could be that the agent arrives at QED. Alternatively, the new state could be an error or represent obligations that are at least as hard as what is currently on the stack (for example, this could be because of a cycle in a tactic). In this case, the agent rejects the new state. Otherwise, it recursively continues the proof from the new state. After issuing a few queries to the LLM, the agent backtracks.

Prompt Serialization Protocol. The routines PROMPTIFY and PARSEACTION together constitute the *prompt serialization protocol* and are critical to the success of the policy. Now we elaborate on these procedures.

PROMPTIFY carefully places the different pieces of information relevant to the proof in the prompt. It also includes logic for trimming this information to fit the most relevant parts in the LLM's context window. Every prompt has two parts: the "system prompt" and the "agent prompt".

The agent prompts are synthetically generated using a context-free grammar and contain information about the state stack (including the current proof state), the textual reward for the previous action, and the set of actions we know to avoid at the current proof state.

The system prompt describes the rules of engagement for the LLM. It contains a grammar (distinct from the one for agent prompts) that we expect the LLMs to follow when it proposes a course of action. The grammar carefully incorporates cases when the response is incomplete because of the LLM's token limits. We parse partial responses to extract the next action using the PARSEACTION routine. PARSEACTION also identifies formatting errors (if any) in the LLM's responses, possibly communicating with the LLM multiple times until these errors are resolved. Figure 5 (in Appendix A.1) shows an example back-and-forth between COPRA and LLM via the prompt serialization protocol.

3 Evaluation

Our findings about COPRA are that: (i) the approach can find proofs significantly quicker than the state-of-the-art finetuning-based baselines, both in terms of number of LLM queries and wall-clock time; (ii) in problems where all current methods fail, COPRA fails faster; (iii) the use of GPT-4, as opposed to GPT-3.5, within the agent is essential for success; and (iv) backtracking significantly improves the system's performance on harder problems. Now we elaborate on our experimental methodology and these results.

Implementing COPRA. The details of our implementation are mentioned in Appendix A.2.1.

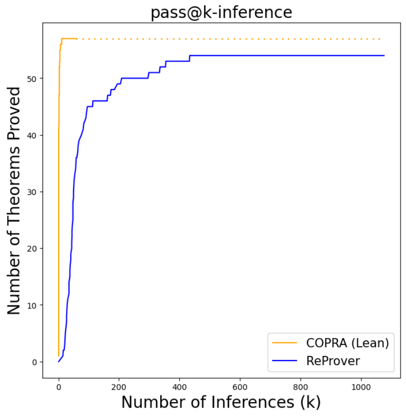


Figure 3: COPRA vs. REPROVER on the `miniF2F` benchmark

Benchmarks. We evaluate our approach on two domains: (i) `miniF2F` (Zheng et al., 2021), a collection of 244 Lean formalizations of mathematics competition problems, solved using a range of techniques such as induction, algebraic manipulation, and contradiction; and (ii) a set of Coq problems from the CompCert compiler verification project (Leroy, 2009) that was previously used to evaluate the PROVERBOT9001 system Sanchez-Stern et al. (2020).

Baselines. We compare with one-shot invocations of GPT-3.5 and GPT-4 in both the `miniF2F` and the Compcert domains. We also consider an ablation of COPRA that uses GPT-3.5 as its LLM and another that does not use backtracking. Our fine-tuned baseline for the `miniF2F` domain is REPROVER, a state-of-the-art open-source prover that is part of the Leandojo project (Yang et al., 2023). In the Compcert domain, we compare with PROVERBOT9001 (Sanchez-Stern et al., 2020), which, while not LLM-based, is the best publicly available model for Coq. More details about the baselines in Appendix A.2.2.

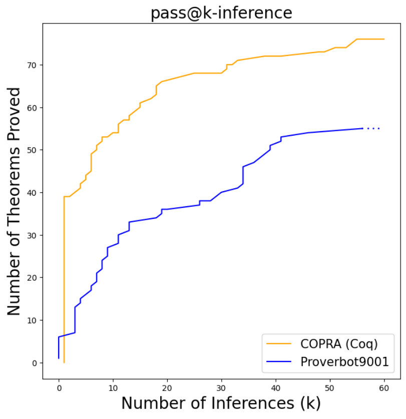


Figure 4: COPRA vs. PROVERBOT9001 on the Compcert benchmark

Metric: pass@k-inferences. The standard metric for evaluating theorem-provers is *pass@k* (Lample et al., 2022; Yang et al., 2023). However, a key objective of our research is to discover proofs *quickly*, with fewer LLM queries and lower wall-clock time. The pass@k metric does not evaluate this characteristic as it does not quantify the number of LLM queries or amount of time needed by a proof attempt.

To address this concern, we introduce a new metric, *pass@k-inferences*, and evaluate COPRA and its competitors using this metric. More details about metric in Appendix A.2.3.

Results Figure 3 and Figure 4 shows that COPRA outperforms the fine-tuned baselines for the `miniF2F` and CompCert domain respectively. We cover more details about results and ablation in Appendix A.3.

References

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.

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.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7): 107–115, 2009.

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*, pp. 218–230, 1957.

OpenAI. Gpt-4 technical report, 2023.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 1–10, 2020.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:9330–9342, 2021.

Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626*, 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.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

A Appendix

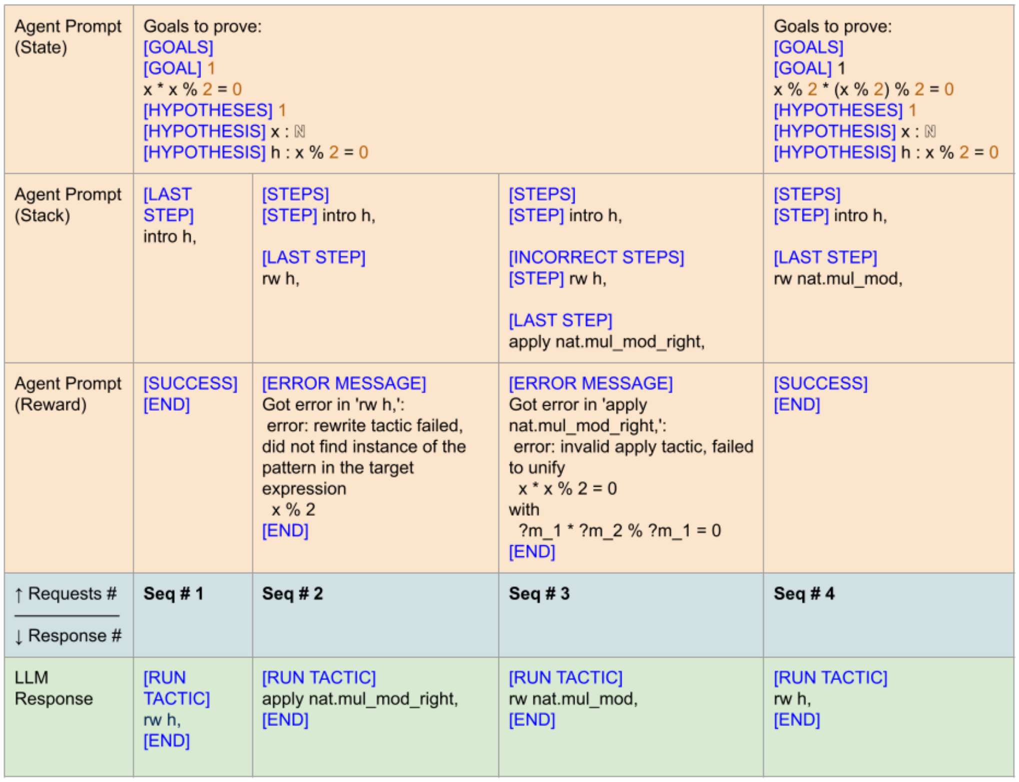
A.1 Prompt Serialization Protocol Example

Figure 5 shows the back-and-forth between the agent and LLM via PSP for a given goal.

A.2 Evaluation Details

A.2.1 Implementation Details of COPRA

Our implementation of COPRA has GPT-4 as the underlying LLM and can interact with both the Lean and the Coq proof environments. Because of the substantial cost of GPT-4 queries, we cap the number of LLM queries that COPRA can make by 60. To further reduce costs, COPRA first tries to



Agent Prompt (State)	Goals to prove: [GOALS] [GOAL] 1 x * x % 2 = 0 [HYPOTHESES] 1 [HYPOTHESIS] x : ℕ [HYPOTHESIS] h : x % 2 = 0			Goals to prove: [GOALS] [GOAL] 1 x % 2 * (x % 2) % 2 = 0 [HYPOTHESES] 1 [HYPOTHESIS] x : ℕ [HYPOTHESIS] h : x % 2 = 0
Agent Prompt (Stack)	[LAST STEP] intro h,	[STEPS] [STEP] intro h, [LAST STEP] rw h,	[STEPS] [STEP] intro h, [INCORRECT STEPS] [STEP] rw h, [LAST STEP] apply nat.mul_mod_right,	[STEPS] [STEP] intro h, [LAST STEP] rw nat.mul_mod,
Agent Prompt (Reward)	[SUCCESS] [END]	[ERROR MESSAGE] Got error in 'rw h,': error: rewrite tactic failed, did not find instance of the pattern in the target expression x % 2 [END]	[ERROR MESSAGE] Got error in 'apply nat.mul_mod_right,': error: invalid apply tactic, failed to unify x * x % 2 = 0 with ?m_1 * ?m_2 % ?m_1 = 0 [END]	[SUCCESS] [END]
↑ Requests # ↓ Response #	**Seq # 1**	**Seq # 2**	**Seq # 3**	**Seq # 4**
LLM Response	[RUN TACTIC] rw h, [END]	[RUN TACTIC] apply nat.mul_mod_right, [END]	[RUN TACTIC] rw nat.mul_mod, [END]	[RUN TACTIC] rw h, [END]

Figure 5: The prompt serialization protocol. We highlight the different parts of the prompts to show how we use the state stack and the textual reward from the environment.

prove its theorems via a single LLM query (one-shot prompting). It only invokes its agent behavior when the one-shot prompting fails to find a proof.

The "system prompt" in the one-shot approach is slightly different than that for COPRA, containing instructions to generate a proof in one go rather than step by step. For both COPRA and the one-shot baselines, the prompt contains a single proof example that clarifies how proofs need to be formatted. This proof example remains the same for all test cases.

A.2.2 Baseline Details

A challenge with the REPROVER baseline is that like COPRA, it uses a retrieval mechanism. However, building a comparable retriever for COPRA would require an indexed training corpus on problems relevant to `miniF2F`. However, `miniF2F` is only an evaluation set and does not come with a training corpus. As a result, for an apples-to-apples comparison, our evaluation on `miniF2F` turns off COPRA's and REPROVER's retrievers.

In the Compcert domain, we compare with PROVERBOT9001 (Sanchez-Stern et al., 2020), which, while not LLM-based, is the best publicly available model for Coq. Unlike `miniF2F`, this benchmark comes with a large training set as well as a test set, and we use the training set for retrieving relevant lemmas and definitions. Our retrieval mechanism, in this case, is a simple BM25 search.

For cost reasons, our evaluation for Compcert uses 118 out the 501 theorems used in the original evaluation of PROVERBOT9001 Sanchez-Stern et al. (2020). For fairness, we include all the 98 theorems proved by PROVERBOT9001 in our subset. The remaining theorems are randomly sampled.

A.2.3 Metric: pass@k-inferences.

The standard metric for evaluating theorem-provers is *pass@k* (Lample et al., 2022; Yang et al., 2023). In this metric, a prover is given a budget of k *proof attempts*; the method is considered successful if one of these attempts leads to success. However, a key objective of our research is to discover proofs *quickly*, with fewer LLM queries and lower wall-clock time. The pass@k metric does not evaluate

Approach	# Theorems proved /# Theorems	% proved	Avg. Inferences in Total	Avg. Inferences on Failure	Avg. Inferences on Pass
miniF2F Test Dataset					
GPT 3.5 Few Shot	7/244	2.8%	1	1	1
GPT 4 Few Shot	26/244	10.6%	1	1	1
COPRA (GPT-3.5)	29/244	11.89%	12.83	14.23	2.45
ReProver	54/244	22.13%	350.7	427.24	81.6
COPRA (GPT-4)	**57/244**	**23.36%**	**20.94**	**26.79**	**1.75**
CompCert Test Dataset					
GPT 3.5 One-Shot	10/118	8.47%	1	1	1
GPT 4 One-Shot	36/118	30.51%	1	1	1
Proverbot	**98/118**	**83.05%**	184.7	256.8	170.0
COPRA	76/118	64.41%	**12.9**	**10.9**	**16.57**

Table 1: Aggregate statistics for COPRA and the baselines on `miniF2F` and Compcert

Approach	Avg. Time In Seconds					
	Per Proof			Per Inference		
	On Pass	On Fail	All	On Pass	On Fail	All
ReProver (on CPU)	279.19	618.97	543.78	3.42	1.45	1.55
ReProver (on GPU)	267.94	601.35	520.74	2.06	0.44	0.48
COPRA (GPT-3.5)	39.13	**134.26**	**122.21**	15.97	9.43	9.53
COPRA (GPT-4)	**30.21**	191.73	140.86	17.26	7.16	6.73

Table 2: Average time taken by our approach (COPRA) and ReProver on miniF2F dataset.

this characteristic as it does not quantify the number of LLM queries or amount of time needed by a proof attempt.

To address this concern, we introduce a new metric, *pass@k-inferences*, and evaluate COPRA and its competitors using this metric. Here, we measure the number of correct proofs that a prover can generate with a budget of *k or fewer LLM inference queries*. One challenge here is that we want this metric to be correlated number of correct proofs that the prover produces within a wall-clock time budget; however, the cost of an inference query is proportional to the number of responses generated per query. To maintain the correlation between the number of inference queries and wall-clock time, we restrict each inference on LLM to a single response.

A.3 Results

Figure 3 shows our comparison results for the `miniF2F` domain. As we see, COPRA outperforms REPROVER, completing, within just 60 inferences, problems that REPROVER could not solve even after a thousand inferences. This is remarkable given that COPRA is based on a black-box foundation model and REPROVER was fine-tuned for at least a week on a dataset derived from Lean's Mathlib library. For fairness, we ran REPROVER multiple times with 16, 32, and 64 (default) as the maximum number of inferences per proof step. We obtained success rates of 15.9%, 20.1%, and 22.13% in the respective cases and took the best for comparison.

Figure 4 shows a comparison between COPRA and PROVERBOT9001.

We find that COPRA is significantly faster than PROVERBOT9001. Since we put a cap of 60 inferences on COPRA, it cannot prove all the theorems that PROVERBOT9001 eventually proves. However, as shown in the figure, COPRA proves many more theorems than PROVERBOT9001 if only 60 inferences as allowed. Specifically, we prove 77.5% of the proofs found by PROVERBOT9001 in less than 60 steps.

Aggregate statistics for the two approaches, as well as a comparison with the one-shot GPT-3.5 and GPT-4 baselines, appear in Table 1. It is clear from this data that the language-agent approach offers a significant advantage over the one-shot approach. For example, COPRA solves more than twice as many problems as the one-shot GPT-4 baseline, which indicates that it does not just rely on GPT-4 recalling the proof from its memory. Also, the use of GPT-4 as opposed to GPT-3.5 seems essential.

Approach	# Theorems proved /# Theorems	% proved
miniF2F Test Dataset		
COPRA (GPT-4) w/o backtracking	56/244	22.95%
COPRA (GPT-4)	**57/244**	**23.36%**
CompCert Test Dataset		
COPRA (GPT-4) w/o backtracking	52/118	44.06%
COPRA (GPT-4)	**76/118**	**64.41%**

Table 3: Ablation showing the effectiveness of backtracking

```
 theorem algebra_sqineq_at2malt1
(a : ℝ) :
a * (2 - a) ≤ 1 :=
begin
    have h : ∀ (x : ℝ), 0 ≤ (1 - x) ^ 2,
    from λ x, pow_two_nonneg (1 - x),
    calc a * (2 - a)
            = 1 - (1 - a) ^ 2 : by ring
        ... ≤ 1 : sub_le_self _ (h a),
end
```

Figure 6: A theorem in the 'algebra' category that COPRA could prove but REPROVER could not.

We establish the correlation between the number of inferences needed for a proof and wall-clock time in Table 2. Although the average time per inference is higher for COPRA, COPRA still finds proofs almost 9x faster than REPROVER. This can explained by the fact that our search is more effective as it uses 46x fewer inferences than REPROVER. These inference steps not only contain the average time spent on generating responses from LLM but at times have some contribution corresponding to the execution of the tactic on the Lean environment itself.

Table 2 also offers data on when the different approaches report failures. Since REPROVER uses a timeout for all theorems, we also use a timeout of 11 minutes while considering failures in Table 2. The data indicates that COPRA is comparatively better at giving up when the problem is too hard to solve. We also note that less time is spent per inference in case of failure for all approaches.

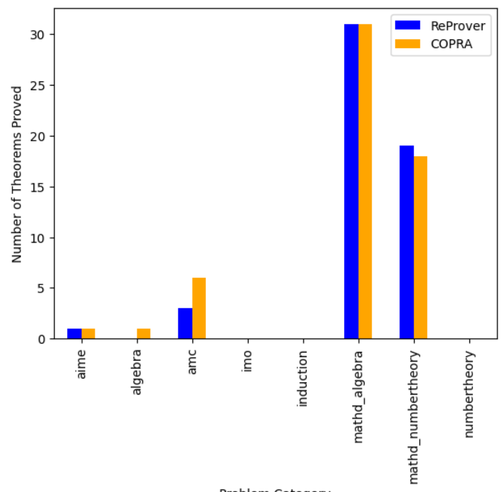
We show the impact of ablating the backtracking feature of COPRA in Table 3. We note that backtracking has a greater positive impact in the Compcert domain. We hypothesize that this is because the Compcert problems are more complex and backtracking helps more when the proofs are longer.

Finally, we offer an analysis of the different categories of `miniF2F` problems solved by COPRA and REPROVER in Figure 7. We see that certain kinds of problems, for example, International Mathematics Olympiad (IMO) problems and theorems that require induction, are difficult for all approaches. However, Figure 7b shows that COPRA takes fewer steps consistently across various categories of problems in `miniF2F`.

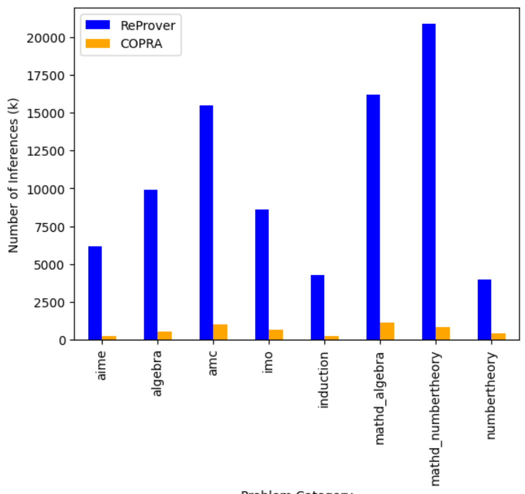
From our qualitative analysis, there are certain kinds of problems where the language-agent approach seems especially helpful. For instance, Figure 6 shows a problem in the 'algebra' category that REPROVER could not solve. More examples of interesting Coq and Lean proofs that COPRA found appear in the appendix.

A.4 Example Proofs generated For miniF2F

Fig. 8 shows some other interesting proofs generated by our approach on miniF2F dataset.



(a) Problems solved in different categories



(b) Number of inferences in different categories

Figure 7: Breakdown of theorems proved in various categories

A.5 System Prompts For miniF2F

Parts of the 'system prompt' used by COPRA for theorem proving in Lean are shown in Fig. 9.

A.6 System Prompts For CompCert

Parts of the 'system prompt' used by COPRA for theorem proving in Coq are shown in Fig. 10.

A.7 Example Proofs generated For CompCert

Fig. 11 shows some interesting proofs generated by our approach on the CompCert dataset.

(a)

```
theorem
    mathd_algebra_246
(a b : ℝ)
(f : ℝ → ℝ)
(h₀ : ∀ x, f x = a * x
    ^4 - b * x^2 + x +
     5)
(h₂ : f (-3) = 2) :
f 3 = 8 :=
begin
    rw h₀,
    rw h₀ at h₂,
    ring_nf,
    linarith,
end
```

(b)

```
theorem
    mathd_algebra_270
(f : ℝ → ℝ)
(h₀ : ∀ x ≠ -2, f x =
     1 / (x + 2)) :
f (f 1) = 3/7 :=
begin
    have h₁ : f 1 = 1
    / (1 + 2),
    apply h₀,
    linarith,
    rw h₁,
    rw h₀,
    field_simp,
    ring,
    norm_num,
end
```

(c)

```
theorem
    mathd_algebra_44
(s t : ℝ)
(h₀ : s = 9 - 2 * t)
(h₁ : t = 3 * s + 1) :
s = 1 ∧ t = 4 :=
begin
    split,
    {
        rw h₀ at h₁,
        linarith
    },
    rw h₀ at h₁,
    rw h₁ at h₀,
    linarith,
end
```

(d)

```
theorem amc12b_2002_p2
(x : ℤ)
(h₀ : x = 4) :
(3 * x - 2) * (4 * x +
     1) - (3 * x - 2)
    * (4 * x) + 1 = 11
     :=
begin
    ring_nf,
    rw h₀,
    ring,
end
```

(e)

```
theorem
    mathd_algebra_107
(x y : ℝ)
(h₀ : x^2 + 8 * x + y
    ^2 - 6 * y = 0) :
(x + 4)^2 + (y-3)^2 =
    5^2 :=
begin
    rw pow_two at h₀,
    rw add_assoc at h₀,

    rw add_comm (x * x
    ) (8 * x + y ^ 2)
    at h₀,
    rw add_comm (8 * x
    ) (y ^ 2) at h₀,
    rw add_assoc at h₀,

    rw add_comm (y ^
    2) (8 * x + x * x)
     at h₀,
    rw add_assoc at h₀,

    ring_nf at h₀,
    rw pow_two,
    rw pow_two,
    ring_nf,
    rw ←add_assoc,
    rw h₀,
    linarith,
end
```

Figure 8: Some other interesting proofs generated for miniF2F by COPRA. The length of the proofs generated shows that interaction with the environment helps in fixing the errors encountered while writing long proofs. These long sequences of rewrites are not easy to synthesize without knowing the exact verbal reward from the environment which often contains the hint to fix the rewrites.

```
You are a proficient formal theorem-proving agent in Lean 3. You can predict
↪  the next proof step given the current proof state. The proof state is
↪  described in the following format:
1. All the goals are described under `[GOALS]` keyword. Each goal within
↪  the `[GOALS]` is described under the keyword `[GOAL] i`, where `i` is a
↪  positive integer. For example, `[GOAL] 1`, `[GOAL] 2`, etc.
2. Within each `[GOAL] i` keyword, the goal is described as a
↪  human-readable serialized version of the proof state as shown while
↪  running `lean` command. Each goal, might also accompany some hypotheses,
↪  which are described under the keyword `[HYPOTHESES] i`. Each hypothesis
↪  within `[HYPOTHESES]`, starts with the prefix `[HYPOTHESIS]`.
3. Sometimes `[GOALS]` can have description about the proof state like
↪  `Proof finished`, `There are unfocused goals`, `Not in proof mode`, etc.
↪  The description is described under the keyword `[DESCRIPTION]`.
4. Finally, `[STEPS]` keyword is used to describe proof-steps used so far.
↪  Each proof step starts with the prefix `[STEP]`, and is a valid Lean
↪  tactic. For example, `[STEPS][STEP]rw h at h,[STEP]{linarith},`.
5. Sometimes, `[INCORRECT STEPS]` keyword optionally used to describe
↪  proof-steps which should NOT be generated. Use this as a hint for not
↪  generating these proof-steps again as they failed previously. For
↪  example, `[INCORRECT STEPS][STEP]apply h,[STEP]rw ←h`.
6. There is also an optional `[LAST STEP]` keyword which describes the
↪  proof-step generated last time. If the proof-step was incorrect, then
↪  it is also followed by error message from Coq environment. For example,
↪  `[LAST STEP]linarith,\n[ERROR MESSAGE]linarith failed to find a
↪  contradiction\nstate:\nx y : ,\nh : x = 3 - 2 * y,\nh : 2 * x - y = 1\n
↪  false`. If the proof-step was correct then it is followed by the
↪  keyword `[SUCCESS]`. For example, `[LAST STEP]linarith,[SUCCESS]`.
↪  Don't generate the last proof-step again if it was NOT successful.
7. Sometimes there can be errors in the format of the generated response.
↪  This is reported using the keyword `[ERROR]` followed by the error
↪  message. For example, `[ERROR]\nInvalid response:\n'Great! The proof is
↪  complete.', \nStopping Reason: 'stop'.\n Please respond only in the
↪  format specified.[END]`. This means that the response generated by you
↪  was not in the specified format. Please follow the specified format
↪  strictly.

If you think you know the next proof step, then start your response with
↪  `[RUN TACTIC]` followed by the next proof-step which will help in
↪  simplifying the current proof state. For example, `[RUN
↪  TACTIC]induction c,[END]`. Generate exactly ONE proof-step. Multiple
↪  proof steps are more error prone, because you will not get a chance to
↪  see intermediate proof state descriptions. Make sure that the proof
↪  step is valid and compiles correctly in Lean 3.

You can refer to the example conversation to understand the response format
↪  better. It might also contain some similar proof states and their
↪  corresponding proof-steps.

 Please take a note of the following:
 1. Make sure to end all your responses with the keyword `[END]`. Follow the
 ↪  specified format strictly.
 2. While generating `[RUN TACTIC]` keyword, do NOT generate the tactics
 ↪  mentioned under `[INCORRECT STEPS]`......
 ..............
```

Figure 9: Parts of ‘system prompt’ used by COPRA for Lean

```
You are a proficient formal theorem-proving agent in Coq. You can predict
↪  the next proof step given the current proof state, relevant definitions,
↪  and some possible useful lemmas/theorems. The proof state is described
↪  in the following format:
1. All the goals are described under `[GOALS]` keyword. Each goal within
↪  the `[GOALS]` is described under the keyword `[GOAL] i`, where `i` is a
↪  positive integer. For example, `[GOAL] 1`, `[GOAL] 2`, etc.
2. Within each `[GOAL] i` keyword, the goal is described as a human-readable
↪  serialized version of the proof state as shown while running `coqtop`
↪  command. Each goal, might also accompany some hypotheses, which are
↪  described under the keyword `[HYPOTHESES] i`. Each hypothesis within
↪  `[HYPOTHESES]`, starts with the prefix `[HYPOTHESIS]`. Apart from the
↪  goal and hypothesis, some OPTIONAL keywords like `[DEFINITIONS] i` and
↪  `[THEOREMS] i` are also present which describe the relevant definitions
↪  of symbols used in that goal, and some possible useful theorems or
↪  lemmas which might help in simplifying the goal. Each definition within
↪  `[DEFINITIONS]` starts with the prefix `[DEFINITION]`. Similarly, each
↪  theorem/lemma under `[THEOREMS]` keyword starts with the prefix
↪  `[THEOREM]`. These definitions and theorems can be used to simplify the
↪  goal using the tactics like rewrite, apply, etc. However, it is also
↪  possible that these definitions and theorems are not used at all.
3. Sometimes `[GOALS]` can have description about the proof state like
↪  `Proof finished`, `There are unfocused goals`, `Not in proof mode`, etc.
↪  The description is described under the keyword `[DESCRIPTION]`.
4. Finally, `[STEPS]` keyword is used to describe proof-steps used so far.
↪  Each proof step starts with the prefix `[STEP]`, and is a valid Coq
↪  tactic ending with a `.`. For example, `[STEPS][STEP]intros
↪  a.[STEP]induction a.`.
5. Sometimes, `[INCORRECT STEPS]` keyword optionally used to describe
↪  proof-steps which should NOT be generated. Use this as a hint for not
↪  generating these proof-steps again as they failed previously. For
↪  example, `[INCORRECT STEPS][STEP]apply mul_assoc.[STEP]rewrite <- H.`.
6. There is also an optional `[LAST STEP]` keyword which describes the
↪  proof-step generated last time. If the proof-step was incorrect, then
↪  it is also followed by error message from Coq environment. For example,
↪  `[LAST STEP]reflexivity.[ERROR MESSAGE]Error: In environment\nn :
↪  nat\nUnable to unify "n" with "n + 0".`. If the proof-step was correct
↪  then it is followed by the keyword `[SUCCESS]`. For example, `[LAST
↪  STEP]reflexivity.[SUCCESS]`. Don't generate the last proof-step again
↪  if it was NOT successful.
7. Sometimes there can be errors in the format of the generated response.
↪  This is reported using the keyword `[ERROR]` followed by the error
↪  message. For example, `[ERROR]\nInvalid response:\n'Great! The proof is
↪  complete.', \nStopping Reason: 'stop'.\n Please respond only in the
↪  format specified.[END]`. This means that the response generated by you
↪  was not in the specified format. Please follow the specified format
↪  strictly.

If you think you know the next proof step, then start your response with
↪  `[RUN TACTIC]` followed by the next proof-step which will help in
↪  simplifying the current proof state. For example, `[RUN TACTIC]destruct
↪  c.[END]`. Generate exactly ONE proof-step. Multiple proof steps are
↪  more error prone, because you will not get a chance to see intermediate
↪  proof state descriptions. Make sure that the proof step is valid and
↪  compiles correctly with Coq.
........................
```

Figure 10: Parts of 'system prompt' used by COPRA for Coq

(a)

```
gss :
forall l v m,
(set l v m) l = match l with R
↪   r =>
    v | S sl ofs ty =>
    Val.load_result
    ↪  (chunk_of_type ty) v
    ↪  end.
Proof.
    intros l v m.
    destruct l as [r | s o t].
    - unfold set.
    destruct (Loc.eq (R r) (R
    ↪  r)); [reflexivity |
    ↪  contradiction].
    - unfold set.
    destruct (Loc.eq (S s o t)
    ↪  (S s o t));
    ↪  [reflexivity |
    ↪  contradiction].
Qed.
```

(b)

```
eq : forall (p q: loc), {p =
↪   q} + {p <> q}.
Proof.
    decide equality.
    - apply mreg_eq.
    - decide equality.
    - decide equality.
    apply Pos.eq_dec.
    decide equality.
    - decide equality.
Qed.
```

(c)

```
disjoint_cons_right
↪  :
forall a l1 l2,
disjoint l1 (a ::
↪  l2) -> disjoint
↪  l1 l2.
Proof.
    intros a l1 l2
    ↪  H.
    unfold
    ↪  disjoint.
    intros x1 x2 H1
    ↪  H2.
    apply H.
    assumption.
    right.
    assumption.
Qed.
```

(d)

```
eq_int_type :
↪  forall (x y:
↪  int_type),
↪  {x=y} + {x<>y}.
Proof.
    decide
    ↪  equality.
Qed.
```

(e)

```
set_locals_lessdef
↪  :  forall e1
↪  e2,
↪  env_lessdef e1
↪  e2 ->  forall
↪  il,
↪  env_lessdef
↪  (set_locals il
↪  e1)
↪  (set_locals il
↪  e2).
Proof.
    intros e1 e2 H.
    induction il as
    ↪  [| a il'].
    - apply H.
    - intros.
    apply
    ↪  set_var_lessdef.
    apply IHil'.
    apply
    ↪  Val.lessdef_refl.
Qed.
```

Figure 11: Some other interesting proofs generated for CompCert by COPRA. We can see that these proofs are long, and often use ‘apply’ tactic which shows that COPRA can effectively use the retrieved information to discharge the current proof state.