Getting More out of Large Language Models for Proofs

Shizhuo Dylan Zhang[1], Emily First[2], and Talia Ringer[1]

[1] University of Illinois Urbana-Champaign, USA
[2] University of Massachusetts Amherst, USA

Abstract

Large language models have the potential to simplify formal theorem proving and make it more accessible. But how to get the most out of these models is still an open question. To answer this question, we take a step back and explore the failure cases of these models using common prompting-based techniques. Our talk will discuss these failure cases and what they can teach us about how to get more out of these models.

Introduction Formal theorem proving is a crucial but challenging task—and a natural fit for automation. Historically, most work on formal proof automation has relied on symbolic techniques [8], sometimes combined with neural tools [2, 7, 12, 6, 4, 3]. Recent advances in large language models have improved their instruction-following and in-context learning capabilities, making it possible to build powerful proof automation that elides symbolic search procedures altogether, as shown by Baldur [5].

How can we get more out of these advances in large language models? To answer that, we examine the capabilities of the state-of-the-art language models GPT-3.5 Turbo and GPT-4 to prove theorems in Coq using common prompting-based techniques. In particular, we conduct a fine-grained analysis of model outputs on an example project. Our emphasis is on the *failure* cases—how these outputs commonly go wrong, and what that can teach us about how to get more out of these models.

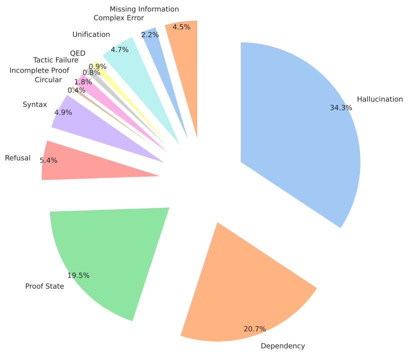
	GPT-3.5-Turbo			**GPT-4**					**Proverbot**
	FS-rand	FS-sim	ZS	FS-rand	FS-sim	ZS	FS+Lem	ZS+Lem	-
#Correct Proof	10	8	0	7	8	0	14	9	-
#Proven Theorems	6	4	0	6	7	0	7	5	23

Table 1: Results. **'FS'** stands for 'few-shot', **'ZS'** stands for 'zero-shot'. **'+lemma'** 'denotes providing lemmas preceding the query theorem in the file in the context.

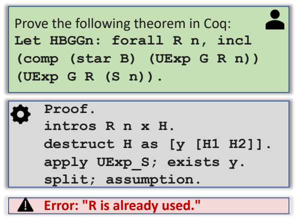
Results and Recommendations We performed fine-grained analysis of the model-generated proofs and categorized the comments of human experts in Figure 1a. Details about our experimental methodology, along with examples, are in the appendix. Our recommendations:

1. Allow the model to prompt the proof assistant for more information. Model outputs sometimes ask for more information about the definitions of referenced variables when those definitions have not been provided as input as shown in Figure 1c and Appendix C.1. This opens up a perfect opportunity for tool use at the model's request. By prompting the model to ask for information using standard Coq commands like `Print`, and executing them in real time, we can allow the model to obtain information as it generates proof in steps.

2. Give the model access to proof states. Language models are found to be weak at 'execution'. In our case, they by default lack access to proof state. This manifests in outputs as incorrect assumptions about the current proof state, like introducing too many variables



(a) Breakdown of 740 expert-annotated GPT-4 outputs under few-shot 'theorem-proof' set-up (see Appendix A.2 for more).



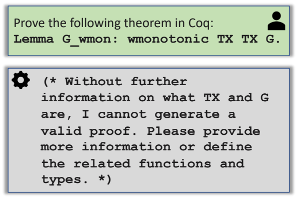
Prove the following theorem in Coq:

```
Let HBGGn: forall R n, incl
(comp (star B) (UExp G R n))
(UExp G R (S n)).
```

```
Proof.
intros R n x H.
destruct H as [y [H1 H2]].
apply UExp_S; exists y.
split; assumption.
```

Error: "R is already used."

(b) An example output that incurs an error because model lacks local proof state, and so uses variable name R that is already taken. See Appendix C.4 for full context.



Prove the following theorem in Coq:

```
Lemma G_wmon: wmonotonic TX TX G.
```

```
(* Without further
information on what TX and G
are, I cannot generate a
valid proof. Please provide
more information or define
the related functions and
types. *)
```

(c) An example output that recognizes the necessary definitions are not provided and asks for clarifications from the user. See Appendix C.1 for more.

or overloading variables already used (see Figure 1b and Appendix C.4). For human proof engineers, interactive proof is more like a conversation with the proof assistant with constant feedback on the state; emulating this is a perfect opportunity to put the chat API to good use.

3. Give the model access to information in file dependencies. Naively prompted models produce outputs that make incorrect assumptions about definitions and lemmas from dependencies, for example by hallucinating lemma names (see Appendix C.6), or by attempting to induct over a non-inductive hypothesis. Human proof engineers avoid this by having access to dependencies directly; models should access to those dependencies in context or in memory.

4. Give the model access to proofs preceding the current proof. Naively prompted models fail to solve proofs that would be obvious from context, since human-written proofs are often small permutations of earlier proofs. Baldur [5] showed in-context learning can help with this in Isabelle/HOL; we now have evidence (see Appendix C.5) this is worth trying in Coq.

5. Learn from errors. People have attempted to iteratively improve the quality of generated programs by providing the model with correctness signals as feedback [13, 10]; the same has shown success for proofs in Isabelle/HOL [5]. The error messages we observed were very informative,[1] and suggest this may be fruitful in Coq as well.

6. Introduce diversity through prompt engineering. It has been observed that a mixture of experts can boost performance of ML-guided proof synthesis by performing diverse sequences of tactics [3]. On the other hand, there is evidence of performance gain by introducing diversity in few-shot set-up for both synthesis [1] and reasoning [11] tasks from the language modeling research community. The diversity we have already observed (see Appendix B) is evidence this may be worth trying with different prompts for Coq proofs.

Proposed Talk In our talk, we will discuss specific examples of these failure cases, how they correspond to our recommendations, and our progress on experimentation along these recommendations in the intervening months.

[1]Our annotated data with errors is available here: https://github.com/DylanZSZ/LLM4Proof.git

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[2] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 09–15 Jun 2019.

[3] Emily First and Yuriy Brun. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 749–761, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Emily First, Yuriy Brun, and Arjun Guha. Tactok: Semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[5] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models, 2023.

[6] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving, 2022.

[7] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[8] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.

[9] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding, 2020.

[10] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback, 2022.

[11] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.

[12] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning, 2021.

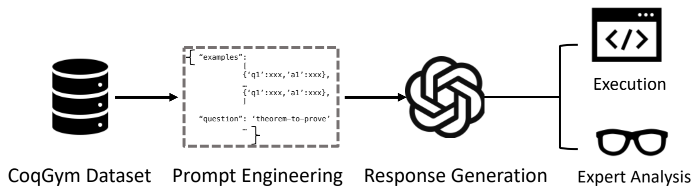
[13] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair, 2023.

A Methodology

We experimented with both zero-shot and few-shot prompting methods with & without preceding lemmas of the query theorem. To select few-shot samples, we experimented with two strategies: random selection and similarity based selection using a retrieval model we fine-tuned on the training set. The details are deferred to appendix.

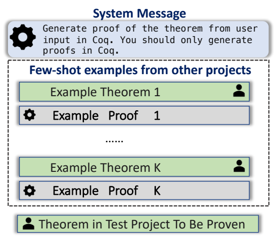
A.1 Zero-Shot Prompting

The notable characteristic of GPT-3.5 Turbo and GPT-4 is their robust ability to follow instructions. In a zero-shot configuration, we leverage the system's messaging to establish the task and desired output format for the entire conversation, while utilizing user messaging to provide query-specific information such as the theorem to be proven and relevant lemmas.



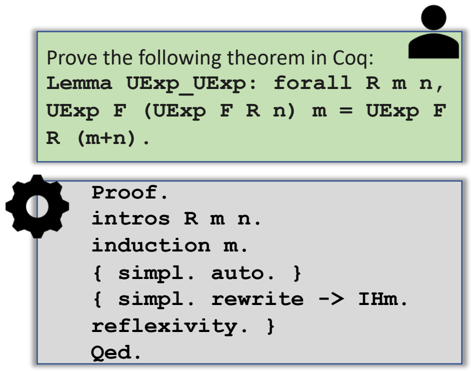
```
{ "examples":
          [
          {'q1':xxx,'a1':xxx},
          …
          {'q1':xxx,'a1':xxx},
          ]

  "question": 'theorem-to-prove'
          … }
```

Figure 2: Investigation Pipeline



```
Generate proof of the theorem from user
input in Coq. You should only generate
proofs in Coq.
```

Figure 3: Workflow of few-shot prompting

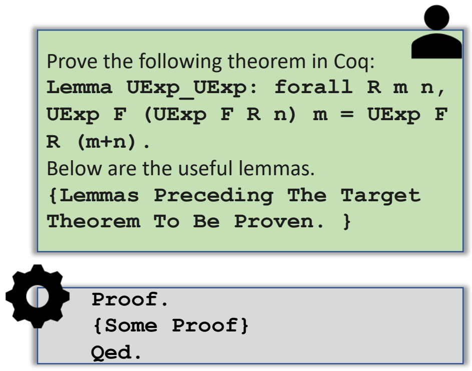


Prove the following theorem in Coq:

```
Lemma UExp_UExp: forall R m n,
UExp F (UExp F R n) m = UExp F
R (m+n).
```

```
Proof.
intros R m n.
induction m.
{ simpl. auto. }
{ simpl. rewrite -> IHm.
reflexivity. }
Qed.
```

Figure 4: Few-shot example format of Theorem-Proof setting.



Prove the following theorem in Coq:

```
Lemma UExp_UExp: forall R m n,
UExp F (UExp F R n) m = UExp F
R (m+n).
```

Below are the useful lemmas.

```
{Lemmas Preceding The Target
Theorem To Be Proven. }
```

```
Proof.
{Some Proof}
Qed.
```

Figure 5: Few-shot example format of Theorem with Given Lemma setting.

A.2 Few-Shot Prompting

Our approach utilizes the conventional few-shot in-context learning method. We include examples in the form of theorem-proof tuples at the beginning of the target query as shown in Figure 3. In addition to the description of the task in the initial system message and the query, we provide examples of query-answer pairs obtained from training set to demonstrate to the model the task we want it to perform and the expected output format. We experimented with a few variants:

Theorem-Proof In this set-up, we provide examples of pairs of theorems and proofs, as illustrated in Figure 4.

Theorem with Given Lemmas In this setting, we provide the model with the names of lemmas preceding the theorem. The format is shown in Figure 5.

A.3 Training of Retriever

We perform continued fine-tuning from the sentence-transformer model **all-mpnet-base-v2** based on[9]. In particular, this model has been trained on code corpus and has certain level of code-understanding capabilities. We train the model to perform similar proof search on the train set by minimizing the triplet objective

$$J = \sum\nolimits_{i=1}^{m} L_{triplet}(T_i, P_i, P_{j \neq i}) \tag{1}$$

where T_i is the query theorem,P_i is the proof of this theorem and P_j is a randomly sampled proof of another theorem. During retrieval, we compute the similarity scores between test theorems and proofs in the training set, ans use those theorems whose proofs are similar to the test instance as the few-shot examples.

A.4 Experiment Set-up

We conducted experiments to evaluate the performance of two language models, GPT-3.5-turbo and GPT4, under both few-shot and zero-shot settings. The few-shot learning was performed with k-shots=6, meaning that the models were trained on only six examples per class.

The hyperparameters used for both models were set as follows: temperature T=1,presence _penalty=0.1, number of samples per prompt n=5. *presence_penalty* is a penalty term applied to the logit values of tokens already present in the prompt to discourage the repetition of those tokens in the generated text.

B Coinciding proofs solved

Table 2: Number of coinciding theorems proven successfully by each pair of prompting set-ups.

	GPT4-random-fs	**GPT3.5-random-fs**	**GPT4-sim-fs**	**GPT3.5-sim-fs**	**GPT4-lemma-fs**	**GPT4-lemma-zs**
GPT4-random-fs	-	-	-		-	-
GPT3.5-random-fs	1	-	-	-	-	-
GPT4-sim-fs	1	2	-	-	-	-
GPT3.5-sim-fs	0	3	1	-	-	-
GPT4-lemma-fs	2	1	1	0	-	-
GPT4-lemma-zs	1	1	1	0	3	-

Proof synthesis is a special task in nature, having similarities with code-synthesis where both being generative, but still different as the criteria of whether or not the target theorem is proved is close-ended to a certain extent.

C Example Outputs

C.1 Refusing To Answer

```
Lemma G_wmon: wmonotonic TX TX G.
  (* Without further information on what TX and G are, I cannot generate a valid
      proof. Please provide more information or define the related functions and
      types. *)
```

Even under few-shot set-up where the model output is restricted to be in Coq format, the model insists on requesting additional information on the variables not clearly defined. It is impressive that the model has certain level of ability to understand Coq as well as the necessary information to complete the proof, e.g. the definition of variable names ans assumptions. This category constitutes 5.4% of total number of cases.

C.2 Not Trying Simple Tactics

```
Lemma trans_incl: incl R R' -> incl (trans R) (trans R').
```

This is an easy theorem which model has failed. In particular, to prove this, a simple auto. tactic will do. However, the model did not give this tactic a try.

```
Lemma trans_incl: incl R R' -> incl (trans R) (trans R').
Proof.
  auto.
Qed.
```

This is a case where all of context, proof state, and dependency information may help. It is also a case where diverse prompting may be useful (for example, by adding a prompt that favors very simple proofs that do not use any lemmas).

C.3 Succeeding

There are certain cases where the model succeeded in proving the theorems. These theorems generally do not depend much on context. For example, the model-generated proof of `weak_refl` succeeds using only simple tactics:

```
Lemma weak_refl: forall x, Weak T x x.
Proof.
  intros x.
  constructor.
  reflexivity.
Qed.
```

C.4 Lacking the Proof State

Here we provide the proof state of the example in the main text before the step in Figure 1b:

```
A, X, Y : Type
TX : reduction_t A X
TY : reduction_t A Y
B : relation X
HB : controlled TX TY B
F, G : function X Y
HF : monotonic TX TY F
HG : wmonotonic TX TY G
HBF : transparent B F
HFG : contains F G
HBG : contains (chaining_l (star B)) G
R : relation2 X Y
HRt : evolve_t TX TY R (comp (star B) (F R))
HRa : evolve_a TX TY R (G R)
pre_silent :
  forall n : nat,
    evolve_t TX TY (UExp F R n) (comp (star B) (UExp F R (S n)))
silent : simulation_t TX TY (comp (star B) (UIter F R))
HFGn : forall n : nat, incl (UExp F R n) (UExp G R n)
______________________________________(1/1)
forall (R0 : relation2 X Y) (n : nat),
  incl (comp (star B) (UExp G R0 n)) (UExp G R0 (S n))
```

Note the proof assistant renamed the `R` in the goal into `R0` because there is another `R` in context from earlier in the file.[2] Coq chooses `R0` so as not to shadow `R`. Similarly, the human proof calls this `RR` to avoid shadowing `R`:

```
intros RR n x y H; right; apply (HBG H).
```

In contrast, without any access to the fact that `R` is already defined the local proof state, the model output attempts to introduce a new variable named `R`. Coq refuses and responds with an error.

Another place where proof state is useful is when the model introduces more variables than can be introduced. For example, the model generates this output:

```
Lemma G_reverse: forall R, eeq (trans (G R)) (G (trans R)).
Proof.
  unfold G, eeq.
  intros R u v.
  destruct R as [R Hr].
  simpl.
  split; intros [r H]; cbn in *; exists r;
  rewrite <- Hr in *;
  auto using sym_equal, trans_sym with relations.
```

The second step, `intros R u v`, assumes there are three variables to introduce. This would be reasonable if unfolding `G` and `eeq` introduced more `forall`s in the goal of the proof state after the first step. But it does not; the local proof state at that step includes only one variable `R` to introduce. Coq thus responds with an error.

C.5 Lacking Local Context

Writing proofs is often easier with context, since proofs often mirror preceding proofs in the same file. For example, proving this lemma is easier with context:

```
Lemma union_incl: (forall i, incl (F i) (F' i)) -> incl (union F) (union F').
```

This is because the human-written proofs preceding it in the file very much mirror the human-written proof of this lemma:

```
Lemma comp_incl: incl R R' -> incl S S' -> incl (comp R S) (comp R' S').
Proof.
  unfold eeq, comp, incl; intuition.
  destruct H1 as [ t ]; exists t; auto.
Qed.

Lemma comp_eeq: eeq R R' -> eeq S S' -> eeq (comp R S) (comp R' S').
Proof.
  unfold eeq, comp, incl; intuition;
  destruct H0 as [ t ]; exists t; auto.
Qed.

Lemma union_incl: (forall i, incl (F i) (F' i)) -> incl (union F) (union F').
Proof.
  unfold eeq, union, incl; intuition.
  destruct H0 as [ i ]; exists i; auto.
Qed.
```

[2]Coq makes it possible to generalize all proofs in a `Section` by the same pre-defined parameters, and one such parameter here is named `R`.

The same holds for the proof of `union2_evolve_right`:

```
Lemma union2_evolve_left:
  forall l R S S', evolve_1 l R S -> evolve_1 l R (union2 S S').
Proof.
  intros l R S S' H x x' y Hxx' xRy; destruct (H _ _ _ Hxx' xRy) as [ y' ];
  exists y'; auto; left; auto.
Qed.

Lemma union2_evolve_right:
  forall l R S S', evolve_1 l R S' -> evolve_1 l R (union2 S S').
Proof.
  intros l R S S' H x x' y Hxx' xRy; destruct (H _ _ _ Hxx' xRy) as [ y' ];
  exists y'; auto; right; auto.
Qed.
```

This gives us evidence that in-context learning in the style of Baldur may be fruitful in Coq as well.

C.6 Hallucination

Below is an example from the model where it hallucinates H which has been defined nowhere. We observe in many cases the model are directly using the commonly used variable names like H, x, y which it has seen multiple times during pre-training.

```
Lemma bisimulation_bisim: bisimulation bisim.
Proof.
  constructor.
  - intros.
    destruct H as [s' H].
    exists s'.
    apply stutter_bisim in H.
    auto.
```

Moreover, it hallucinates lemmas or definitions in the proofs, like `stutter_bisim`. By providing proof state, we can help the model pick more correct variables; by providing file dependencies that contain referenced definitions and auxiliary lemmas, and prompting the model to restrict itself to those, we can help the model use only the definitions and lemmas that exist already.