PVSGym: A Proof Learning Environment

Manoj Acharya, Karthik Nukala, Natarajan Shankar
SRI International
Menlo Park, CA 94025, USA
{manoj.acharya,karthik.nukala,natarajan.shankar}@sri.com

Abstract

We introduce PVSGym, a reinforcement learning (RL) gym environment tailored for the PVS (Prototype Verification System) theorem prover. PVSGym includes an ML-friendly dataset of expert-level PVS formalizations drawn from the NASALib library which span a number of mathematical and computational domains. Our aim is to catalyze research at the intersection of AI and formal methods, ultimately advancing proof automation and mathematical discovery. Both the environment and dataset will be publicly available.

1 Introduction

The advent of large language models (LLMs such as OpenAI's o1 Jaech et al. [2024] and DeepSeek-R1 Guo et al. [2025]) has opened up the possibility of constructing problem solving frameworks that combine soft reasoning (e.g., natural language processing and generation, pattern matching, search) with hard reasoning procedures (e.g., theorem provers, constraint solvers, rewrite engines). LLM-driven AI agents are susceptible to hallucination and flawed reasoning, while formal reasoners tend to be brittle and hard to use. The application of LLMs in assisting with the formalization and proof of mathematical theorems is one such application. The ultimate goal here is to help users express mathematical specifications in a formal language and to train models to generate definitions, conjectures, and proofs in this language.

We introduce PVSGym, the first gym environment for the PVS (Prototype Verification System) theorem prover. PVS intertwines rich dependently-typed specifications and aggressive state-of-the-art proof automation into a coherent specification/verification environment that can be readily extended to fit any desired niche. PVSGym integrates reasoning engines like PVS and its supporting symbolic decision procedures with AI agents to construct formally verified mathematical proofs. Our work advances the broader goal of automated mathematical discovery by bridging advances in large-scale machine learning with the rigor of formal verification.

In summary, our contributions are:

- The first interactive gym-style environment Brockman et al. [2016], PVSGym, for PVS, enabling the training and evaluation of AI agents for proof search.
- A dataset of theorems and their corresponding proofs derived from the NASALib library, a collection of industrial-scale theorems used in the mathematical and computational modeling of safety-critical systems. Each proof is constructed by expert users (NASA engineers, PVS developers) to ensure correctness and diversity, providing high-quality trajectories that can serve both as training data and as evaluation benchmarks.
- An application of PVSGym and NASALib to establish initial baselines for RL-guided theorem proving.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.

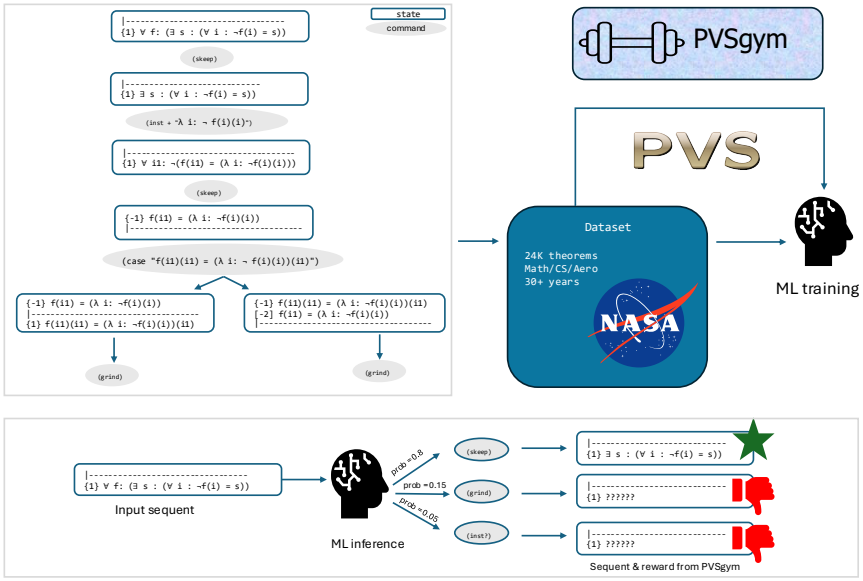

Figure 1: The figure at the top depicts an interactive PVS proof session of Cantor's Theorem. The `NASALib` dataset contains a number of formalizations used to train/finetune the AI agent in a reinforcement-learning loop. The figure at the bottom depicts an inference-time workflow for predicting the next command. `PVSGym` provides both insight into the model's prediction probabilities and an evaluation of the reward by applying the proof commands to the proof state.

1.1 Related Work

For a comprehensive summary of related work, see Yang et al. [2024]. We highlight here two primary sources of inspiration: the CoProver project Yeh et al. [2023] for proof step prediction and lemma retrieval and the LeanDojo project Yang et al. [2023] for an integrated environment for AI/agentic integrations to computed-assisted theorem-proving.

2 PVS: A Prototype Verification System

PVS was initially developed over thirty years ago and has been continuously maintained and improved since then. The PVS specification language is based on higher-order logic extended with dependent function, tuple, and record types, predicate subtypes, algebraic and coalgebraic datatypes, and parametric theories. Type correctness is undecidable, but is decidable modulo proof obligations called Type Correctness Conditions (TCCs). Nearly all of the specification language is executable as a functional language, and there are generators for extracting efficiently executable code in Common Lisp and C (with experimental generators for OCaml and Rust). The type systems makes mathematics coherent by ruling out anomalies such as division by zero or taking the square root of a negative number. It also makes computation safe so that the execution of a well-typed PVS program cannot trigger runtime errors. Many ideas initially prototyped in PVS have been adopted in other proof assistants and verification tools.

Theorem proving, computer-assisted or otherwise, is a practice fraught with *sparse reward signals*. Taking steps is like walking in the dark. The PVS methodology has been to reduce the number of steps you take in the dark and to correct missteps as early as possible. This is accomplished by the tight integration of decision procedures and proof automation in every command. PVS ships with an arsenal of such proof engines like the Shostak decision procedures Shostak [1984] and has been further enhanced with solvers for monadic second-order logic (WS1S Owre and Ruess [2000]), MetiTarski (Denman and Munoz [2014]), binary decision diagrams, symbolic model checking, external SMT solvers (Yices/Z3). PVS provides user-defined hooks to flexibly develop and integrate external oracles, allowing users to define what's obvious in their domain of interest and reason at its appropriate level of abstraction. For instance, NASA has implemented a sub-theorem prover for differential dynamic logic and hybrid programs entirely within PVS Slagel et al. [2024], which is available through our `NASALib` distribution.

```
sum: THEORY
 BEGIN

 n: VAR nat

 sum(n): RECURSIVE nat =
  (IF n = 0
   THEN 0
   ELSE n + sum(n - 1)
   ENDIF)
  MEASURE n

 closed_form: THEOREM
    sum(n) = (n * (n + 1))/2

 END sum
```

2.1 PVS specification file (`.pvs`) with formula declarations and theorem statements.

```
(induct-and-simplify "n")
```

2.2 A combined PVS proof command applying induction on `n` and applying aggressive simplification.

```
closed_form :
|-------
{1}  FORALL (n: nat):
     sum(n) = (n * (n + 1)) / 2
```

2.3 PVS sequent on which proof commands will be applied.

Figure 2: A simple PVS theory formalizing the nth triangular number $\Sigma_{i=0}^{n} i$, and a proof relating it to its closed-form solution. The PVS tactic `induct-and-simplify` invokes a high-level induction step and dispatches low-level arithmetic/theory reasoning with decision procedures and proof engines.

```python
async def run_agent(PORT, file_path, formula_name):
  async with PVSExecutor("localhost", str(PORT)) as env:
    sequents, info = await env.reset(file_path, formula_name)
    print(f"Goal: {sequents}")
    done = False
    while not done:
      ## recieve command from the agent
      state, reward, terminated, truncated, info = await env.step(command)
      done = terminated or truncated
```

Figure 3: `PVSGym` parses and typechecks the PVS specification and then enters the PVS prover awaiting user or agent commands. Useful information regarding proof state, reward for executed command, and debug information is readily provided through the API.

3 PVSGym

`PVSGym` is an interactive environment for PVS similar to the OpenAI Gym Brockman et al. [2016] interface, enabling programmatic interaction. To the best of our knowledge, no prior framework exists that allows users or agents to interact with PVS in a tactic-level execution loop. This environment transforms PVS into a symbolic reasoning engine, where agents can issue commands and receive structured feedback from the PVS prover. In addition to interactive capabilities, the `PVSGym` environment supports command communication via a JSON-RPC API to evaluate multiple proof tree expansions. Together these features are valuable for large-scale data generation, high-throughput training, and evaluation of proof-generating agents.

`PVSGym` accepts a PVS specification file and parses/typechecks the specification. The system then returns a goal sequent for a provided theorem, as shown in Figure 3. After that, the user can send commands and receive the updated proof state after each command is executed. `PVSGym` also supports custom reward functions defined on state/action pairs. For example, it includes a default reward function that mimics human provers:

- High rewards if all branches are proved
- Smaller positive rewards if some branches are proved or partial progress is made
- Negative rewards if a bad command is issued or if the state does not change after applying a command

4 Experiments

`NASALib` **Dataset:** We construct a large dataset of proof rollouts extracted from the NASA PVS Libraries called `NASALib` [LaRC]. `NASALib` is a comprehensive formal methods library consisting of 62 top-level libraries with around 38K proven formulas (including TCCs), designed to support a wide range of mathematical, logical, and application-specific domains. It spans foundational areas such as algebra, analysis, topology, linear algebra, probability, and number theory, while also covering specialized topics like differential dynamic logic, interval arithmetic, multivariate polynomials, term rewriting systems, cooperative game theory, and aviation safety. The library integrates automated reasoning tools like MetiTarski, provides executable specifications for structures like matrices and graphs, and formalizes advanced areas including Lebesgue and Riemann integration, ordinary differential equations, and temporal logics. The full library and its dependencies can be seen in Figure 5 in the Appendix.

From these proofs we extract definitions, lemmas, and step-by-step command sequences required to complete the proofs, including both successful tactic applications and intermediate sequents. The resulting dataset comprises over 21,743 proof trajectories, each representing a grounded reasoning task in a formal domain.

Command Prediction Task: We frame theorem proving as a command prediction task, where the model is trained to generate a valid PVS command given an existing proof state and supporting material (prior proof states and theory declarations). We experiment with three input configurations: (i) command-only, (iii) sequent-only (iii) command + current sequent, and all results are reported in Table 1.

Evaluation: We benchmark on a 95%/5% train-test split with the test set constructed from files and libraries that are completely disjoint from those used during training. This ensures that no proof steps, definitions, or contextual information from the test set appear during training. By isolating the source files, we prevent overlap at both the example and contextual knowledge levels, providing a more realistic assessment of model performance on unseen proofs.

Experimental details: We fine-tune CodeLlama-7B Roziere et al. [2023] on our rollout dataset with an autoregressive next-token prediction objective. For example, for the cmdhist+sequent setting, the model is prompted with the current sequent and the previous *k=3* commands, and trained to predict the next command. Training on a single NVIDIA A100 GPU takes 10–12 hours.

During inference, we apply temperature sampling to generate top-k candidate predictions. Each candidate is scored based on its average log probability, and for every generated output we extract only the first line (split by newline characters) as the predicted command. This approach ensures both consistent formatting and alignment with the expected PVS command syntax.

5 Results

Table 1 shows that predictive accuracy improves as the model is given richer contextual information. Using only command history performs worse than using only the current sequent, while combining both yields the best performance. This highlights the complementary nature of structural (sequent) and procedural (history) signals for guiding proof search. Prior work such as CoProver Yeh et al. [2023] has explored similar models but is limited to command-type prediction (e.g., `(assert)`, `(expand)`, `(inst)`, `(replace)`), without addressing the more challenging task of predicting full commands with arguments.

We also present the confusion matrix of our command prediction model on a held-out test set in the Appendix Figure 4. While some frequently used commands such as `(assert)`, `(expand)`, and `(inst)` are predicted with high accuracy, while syntactically or semantically similar commands (e.g. `(skolem)`/`(skosimp)`/`(skeep)`) exhibit moderate confusion. A more detailed analysis of error patterns is provided in the Appendix.

Table 1: Command prediction accuracies, with and without command history information. Note that CoProver* Yeh et al. [2023] evaluates only the command-type prediction task, without testing the prediction of the correct command arguments.

Method	**cmdhist + sequent**	**sequent**	**cmdhist**
K-NN Yeh et al. [2023]	0.28	0.19	0.27
CoProver* Yeh et al. [2023]	0.48	0.28	0.21
Ours	0.49	0.47	0.42

6 Conclusion

We introduced `PVSGym`, the first interactive environment for the PVS theorem prover, together with an expert-vetted dataset of proof rollouts from the industrially used `NASALib`. Both resources, along with baselines, documentation, and reproducible scripts will be released publicly. We hope this lowers the barrier to entry for researchers and practitioners of the PVS system, and provides a foundation for advancing AI-assisted theorem proving.

References

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

W. Denman and C. Munoz. Automated real proving in pvs via metitarski. In *International Symposium on Formal Methods*, pages 194–199. Springer, 2014.

D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

N. L. R. C. (LaRC). NASALib: PVS NASA library, 2020. URL `https://github.com/nasa/pvslib`.

S. Owre and H. Ruess. Integrating ws1s with pvs. In *International Conference on Computer Aided Verification*, pages 548–551. Springer, 2000.

B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

R. E. Shostak. Deciding combinations of theories. *Journal of the ACM (JACM)*, 31(1):1–12, 1984.

J. T. Slagel, M. Moscato, L. White, C. A. Muñoz, S. Balachandran, and A. Dutle. Embedding differential dynamic logic in pvs. *arXiv preprint arXiv:2404.15214*, 2024.

K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

K. Yang, G. Poesia, J. He, W. Li, K. Lauter, S. Chaudhuri, and D. Song. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024.

E. Yeh, B. Hitaj, S. Owre, M. Quemener, and N. Shankar. Coprover: a recommender system for proof construction. In *International Conference on Intelligent Computer Mathematics*, pages 237–251. Springer, 2023.

A Appendix

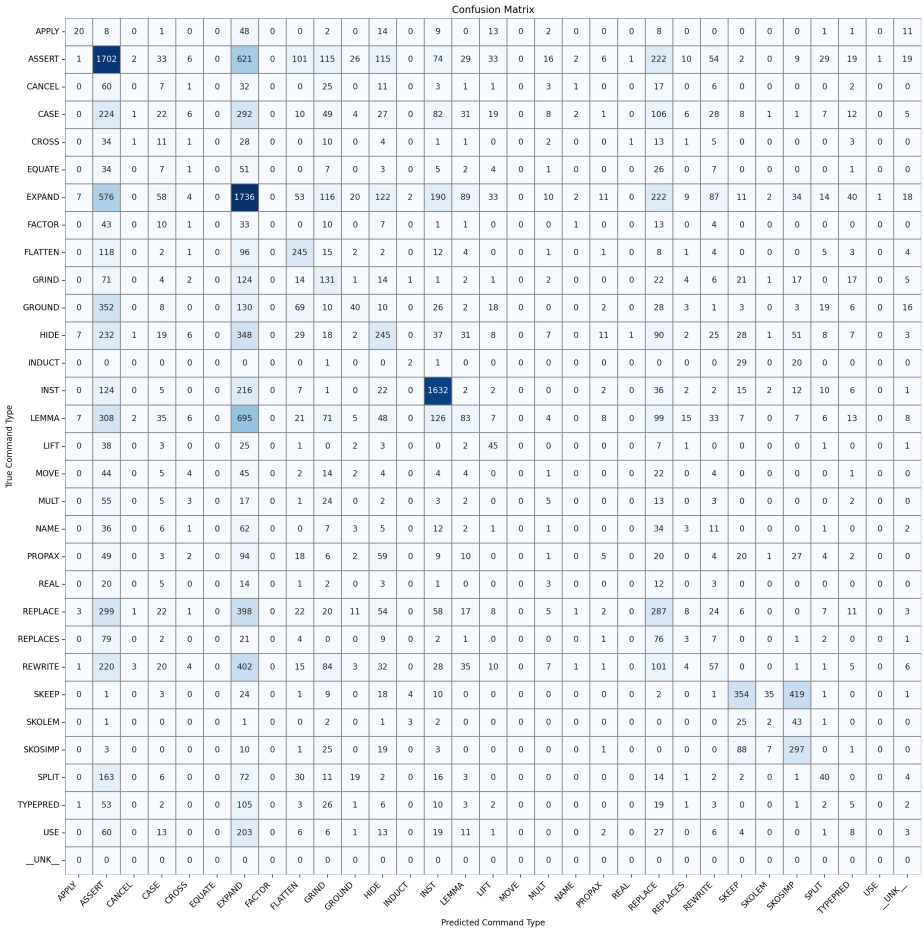

Figure 4: Confusion matrix

We present a confusion matrix of our command prediction model on a held-out test set Figure 4. Notably, some commands are highly represented such as `(assert)`, `(expand)`, `(inst)` with each showing over 1,000 correct predictions. Some syntactically similar commands such as `(replace)`, `(replaces)`, and `(rewrite)` are often confused with one another, likely due to overlapping lexical patterns and similar usage contexts. Similarly, tactical proximity for commands like `(skolem)`, `(skosimp)`, and `(skeep)` also exhibit moderate off-diagonal confusion, suggesting that semantically related tactics are harder to disambiguate without richer context. Finally for low-frequency commands which occur rarely in the training dataset such as `(induct)`, `(move)`, or `(typepred)` tend to show weaker prediction accuracy and higher dispersion across predicted labels.

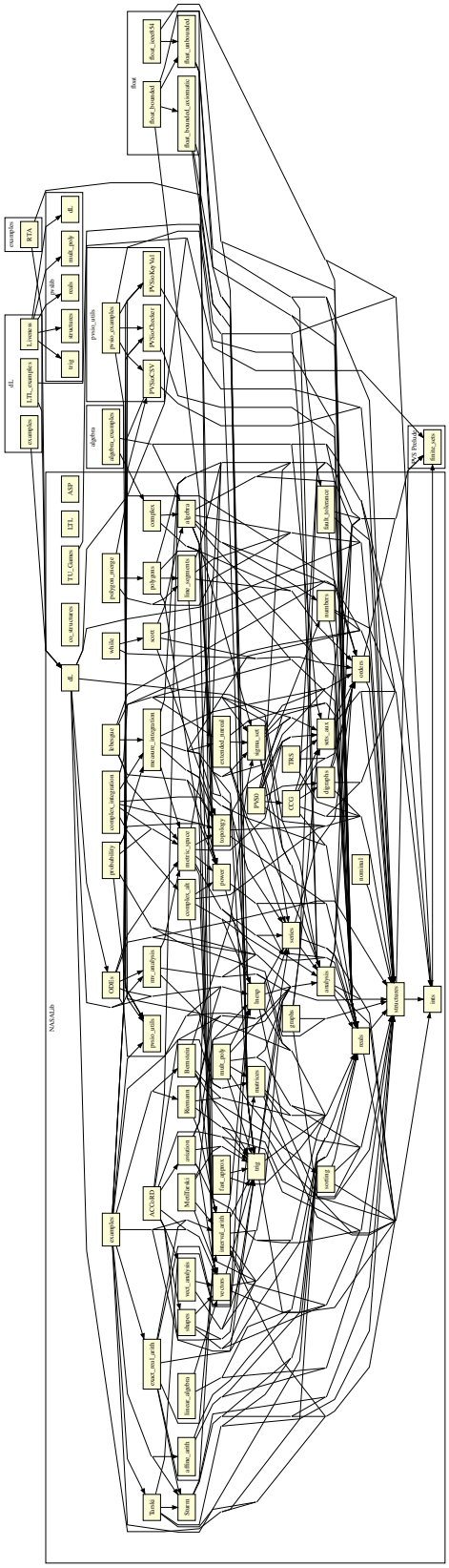

Figure 5: Summary of `NASALib`.