Automating Interactive Theorem Proving

Justin Restivo
justin.restivo@yale.edu

Abstract— Preventing software bugs has always been an important problem in software engineering largely due to the usage context. For example, bugs in both mission critical software and large scale deployment settings can be disastrous. We present a literature review of the current research on preventing bugs in an automated fashion. Then, we discuss three current works that apply machine learning to solve this problem: LegoProver, Curriculum Learning, and LeanDojo. Finally, we provide a history of prior approaches to bug prevention.

I. The Task

The task is to prevent software bugs before software is deployed. Software is used in many important contexts in which bugs can be catastrophic. Real world examples include self driving cars, nuclear [1], medical devices [2], and financial systems [3] . Furthermore, software bugs are becoming worse with time. With introduction of AI assisted coding practices such as Copilot [4] or Codeium [5], studies find that software code quality has gone down [6], Which increases the liklihood of bugs.

II. Task History and Related Work

There have been many approaches and decades of research devoted to solving the problem of writing correct, bug-free software. We mention several of the most popular approaches.

A. Programming Languages

According to Microsoft, 70% of their bugs were due to errors in manual memory management (denoted memory safety) [7] . One part of the solution has been to write application code in memory safe langauges, like Rust, Golang, Java, or Python. Even the US government recommends the use of memory safe languages to prevent bugs [8] .

Other common causes of bugs include implicit type coersion and runtime type errors. For example, runtime type errors became so common that languages that allow runtime type errors introduced types. Notably, Javascript has largely been replaced by Typescript. And, Python has introduced optional type annotations and a type checker.

Type systems serve as a bandaid by using the compiler to prevent bugs. However, this does not solve the entire problem. There still remains a class of logical bugs that are not caught by the compiler. These bugs must be dealt with in other ways.

B. Automated Theorem Proving

One way to guarantee correct code is to specify the utility of each function. This specification is generally created via pre and post conditions for each function (as well as loop invariants). Then, the function's code can be verified to match the specification. This may be done by converting the code to guarded commands [9] then creating a formula from those commands using hoare logic. If the negation of that formula is satisfiable, then there exists an input such that the specification is violated. That is, a bug has been identified.

A popular implementation of this methodology is Dafny [10] . However, there are still several open problems with this approach. Checking if the formula is satisfiable is a NP-complete problem. So, verifying the code may run for an unbounded and unpredictable period of time. One solution to this is to run several SAT solvers with different techniques/flags at the same time (and possibly propagate information between them) [11], [12] . However, this is an imperfect solution because of the higher compute demands for a potential speedup.

Another downside is the need to specify loop invariants in order for the hoare logic to be applicable. Large amounts of compute is required. Additionally, reasoning about parallel programs is difficult.

C. Interactive Theorem Proving

Another prominent approach is to use an interactive theorem prover such as Lean or Coq. Interactive theorem provers are able to, using logical axioms, work with a programmer to interactively prove facts about code. For example, a programmer could define a palindrome, then prove that the reverse of the palindrome is also a palindrome [13] . This scales better from a computation standpoint, since NP-complete problems need not be solved to verify the code. On the other hand, this approach is time consuming in programmer hours.

Large scale examples of this approach include:

- CompCert, a verified C compiler, which is 30k lines of ocaml code and took 6 person years to verify [14]
- Certikos, a verified kernel, which is 6.5k lines of C/asm code and took 3 person years to verify [15]

- SEL4, a verified kernel, which is 9k lines of C/asm code and took 11 person years to verify [16]

III. Interactive Theorem Proving with Machine Learning

If there were a way to automate the theorem proving, then the programmer would not need to spend time proving the code. This would make interactive theorem proving much more feasible, thereby making verified code much more common. Although a relatively new field, we survey recent contributions in three recent works: Curriculum Learning, LeanDojo, and LEGO-Prover.

A. Curriculum Learning

Curriculum Learning's primary contribution is the application of expert iteration [17] to automate the proof generation for theorems in Lean. Though the original usecase for this technique was chess, Karpukhin et al. [17] applies the same techniques to iteractive theorem proving because of the similarities in search space size and game type.

Karpukhin et al. started with GPT-3 pretrained on Commoncrawl (a web dataset) and webmath (a math dataset). They then defined two objective functions. The ProofStep objective function used the theorem name as a heuristic to enforce recall of related data. And the ProofSize objective function used the estimated size of the proof to prioritize smaller proofs. The model is then trained iteratively starting with the aforementioned pretrained model, these objective functions, and a tree search over the application of different tactics (branches) to different states (goals to prove). This model may then be used in conjunction with a best-first search using a heuristic called logprob [18] to determine which branch to traverse.

This approach worked reasonablely well, as it was able to prove 36.6% of the theroems in the MiniF2F test benchmark [19] . However, it used LeanStep for data extraction and contributed lean-gym for data training, respectively. These two frameworks have since gone unmaintained.

B. LeanDojo

LeanDojo reimplemented data extraction and forked lean-gym, both improving these libraries and upgrading them to handle Lean 4. LeanDojo's second contribution was a computationally cheap to train and run machine learning guided interactive theorem prover, denoted ReProver.

ReProver reuses the insights developed by Karpukhin et al. both of using a large language model to generate tactics to progress through a proof and a best-first search using logprob. ReProver also emphasized the importance of choosing the right premises (known facts) as arguments to the tactic generation. ReProver applied the idea of dense passage retrieval [20] to select related premises to the state to prove. It then fed these premises and the state into a large language model to generate the tactics to use in the best-first search.

This approach seemed to work really well given the limited resources of the model, proving 26.5% of theorems in MiniF2F. LeanDojo's large language model has an order of magnitude less parameters than Karpukhin et al. and emphasized short training times on consumer-level hardware.

LeanDojo also contributed a large dataset including training data (a large improvement over MiniF2F).

C. Lego-Prover

Lego-Prover solves a slightly different, but related problem. Lego-Prover takes a paper proof of the theorem, a english version of the theorem in addition to premises and the theorem itself. Using the paper proof, Lego-Prover then splits the theorem into the composition of simpler lemmas. Lego-Prover uses K nearest neighbors to select premises to feed into the llm to prove the lemmas. Lego-Prover constantly adds to this lemma database, and improves each lemma (denoted "evolution") with a large langauge model before choosing the lemmas to use in progressing the state. Notably, unlike LeanDojo and Karpukhin et al. Lego-Prover does not use a best-first search approach.

Lego-Prover performed better than ReProver or Karpukhin et al. with a 50% success rate on MiniF2F. However, the large language model used was GPT4, which is significantly larger than both LeanDojo and Reprover. So, better performance is not unexpected due to a higher parameter count.

IV. Conclusion

Bugs in code are a serious problem, and researchers have been working for decades on their prevention to varying levels of success. Common bug prevention techniques include automated theorem proving, better type systems, and interactive theorem proving. However, only recently has the approach of interactive theorem proving begun to become automated using machine learning techniques. LeanDojo, LEGO-Prover, and Curriculum Learning all provide encouraging results that in the future may make writing bug free code easier.

References

[1] M. Baezner and P. Robin, "Stuxnet", 2017.

[2] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents", *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[3] W. Li, J. Bu, X. Li, H. Peng, Y. Niu, and Y. Zhang, "A survey of DeFi security: Challenges and opportunities", *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10, pp. 10378–10404, 2022.

[4] “Github Copilot”. [Online]. Available: https://github.com/features/copilot

[5] “Codeium”. [Online]. Available: https://codeium.com/

[6] “Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality”. [Online]. Available: https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality

[7] “Github Copilot”. [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[8] “Github Copilot”. [Online]. Available: https://media.defense.gov/2023/Dec/06/2003352724/-1/-1/0/THE-CASE-FOR-MEMORY-SAFE-ROADMAPS-TLP-CLEAR.PDF

[9] E. W. Dijkstra, *A Discipline of Programming*, 1st ed. USA: Prentice Hall PTR, 1997.

[10] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness”, in *International conference on logic for programming artificial intelligence and reasoning*, 2010, pp. 348–370.

[11] G. Audemard, B. Hoessen, S. Jabbour, and C. Piette, “Dolius: A Distributed Parallel SAT Solving Framework.”, in *POS@ SAT*, 2014, pp. 1–11.

[12] J. Burchard, T. Schubert, and B. Becker, “Distributed parallel# SAT solving”, in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 326–335.

[13] “Codeium”. [Online]. Available: https://lean-lang.org/lean4/doc/examples/palindromes.lean.html

[14] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, “CompCert-a formally verified optimizing compiler”, in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[15] R. Gu *et al.*, “{CertiKOS}: An Extensible Architecture for Building Certified Concurrent {OS} Kernels”, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 653–669.

[16] G. Klein *et al.*, “seL4: Formal verification of an OS kernel”, in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[17] D. Silver *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”, *arXiv preprint arXiv:1712.01815*, 2017.

[18] S. Polu and I. Sutskever, “Generative language modeling for automated theorem proving”, *arXiv preprint arXiv:2009.03393*, 2020.

[19] “Github Copilot”. [Online]. Available: https://github.com/openai/miniF2F

[20] V. Karpukhin *et al.*, “Dense passage retrieval for open-domain question answering”, *arXiv preprint arXiv:2004.04906*, 2020.