Reflection for
Systematic Benchmarking Test Case Generation

Team 1

Emma Willson
Esha Pisharody
Grace Croome
Marie Hollington
Proyetei Akanda
Zainab Abdulsada

April 4, 2025

GitHub Repository

https://github.com/proyetei/CS-4ZP6A-Capstone-Akanda

Achievements

Through this project, we built a benchmarking framework for proof assistants by designing a shared internal grammar and implementing pretty-printing translators to generate equivalent code across the four languages (Agda, Idris, Lean and Rocq). We developed 21 scalable, parameterized tests to evaluate their time and memory performance.
To streamline workflows, Docker and GitHub Actions were used. These CI workflows supported the generation of tests by size with a specified range or a list of sizes, ensuring reproducibility and scalability with minimal effort.
Overall, we were able to get a variety of interesting results showcasing performance metrics.

Requirements

All P0 and P1 requirements were met. Rather than having the project as a single downloadable package, it has been set up in Docker, hence the user needs to pull the Docker image to set up the project locally.

P0

- The system will contain a grammar MHPG
- The system will contain a hard-coded initial set of EPs written in our proof grammar (MHPG).
- The system will be given a base set of EPs of various types (propositional, natural induction/deduction).
- The system, when given a number of operations and a test from the initial set, will increase the size of the given test case by the provided number of operations.
- The system will generate a set of tests of increasing sizes from the initial set.
- Post-test case generation, the system will translate them into the four PALs.
- The system will compile the test cases in each PAL.
- The system will record and display the compilation time of each of these test cases in the four PALs.
- The system will provide documentation on how to extend the test generator with new classes of tests.
- The system will support installation on UNIX environments.

P1

- The system will measure the time and memory complexity of each test case as determined by comparison of the same base case across increasing sizes.
- The system will visualize the measured data in graphs on multiple webpages.
- The system will provide a description for each graph displayed on the webpages, explaining the data being visualized.
- The system will be available as a single downloadable package that enables users to install the complete system.

What Didn’t Work

Developing BenchPAL presented several technical and organizational challenges, particularly when dealing with Windows environments. One of the primary obstacles was ensuring compatibility across different operating systems, as Docker—a key component for

containerized deployment—proved unreliable on Windows. Additionally, installing the four PALs (Lean, Idris, Agda, and Rocq) on Windows machines was significantly less reliable than on Unix-based systems.

During deployment, we initially attempted to use GitHub Pages integrated with a Flask backend, but this approach failed due to compatibility and hosting constraints. After evaluating alternatives, we transitioned to Vercel, which provided a more seamless deployment pipeline. One limitation we discovered was Lean's unexpected memory constraints, which required error-handling to prevent crashes during testing. We initially considered implementing a Bash script to run our testing system, but this was abandoned due to cross-platform complications.

Beyond technical difficulties, we recognized the importance of workload distribution in team collaboration. One team member bore a disproportionate burden in setting up the entire continuous integration (CI) framework, highlighting the need for a more balanced division of tasks. Moving forward, a clearer assessment of individual workloads and dependencies would improve efficiency and reduce personal strain in similar projects.

What Worked

One of the key factors in our success was our effective team coordination and technical implementation. The division of work was well-structured, with clearly segregated responsibilities that minimized GitHub merge conflicts and allowed for parallel development without significant integration hurdles.

Planning played a crucial role in maintaining productivity. Esha's suggestion to hold longer, more focused meetings proved invaluable, as it allowed us to retain context and dive deeper into complex issues without constant reorientation. Communication was streamlined through Discord, enabling rapid decision-making and quick clarifications with our supervisor when needed. Additionally, biweekly meetings with our supervisors provided consistent feedback and helped align our progress with expectations.

On the technical side, our use of GitHub Issues for task tracking ensured accountability and transparency, making it easy to monitor progress and redistribute work when necessary. The decision to implement the command-line interface in Go was particularly beneficial, as its simplicity, performance, and cross-platform compatibility reduced development friction. While we encountered challenges with certain components, the majority of our tooling and processes worked as intended, contributing to a stable and maintainable system.

What We Learned

From both our successes and challenges, we learned that clear task division and efficient communication are crucial for smooth collaboration, while platform compatibility and workload balance require careful planning from the outset. Additionally, we saw firsthand how flexible tooling choices (like Go for CLI) and adaptive processes (like longer meetings) can significantly improve outcomes, whereas assumptions about cross-platform support (like Docker on Windows) need rigorous early validation.