Haskudoku report (Functional Programming Project)

(haskudoku = haskell + sudoku, by Vladimir Lezhnev)

First of all, how to run it?.

Run site.html in your browser and run “stack run” in your terminal. Tests - “stack test”

What the task was.

The task was to implement performant Sudoku solver and generator (which we will later here realize actually the same thing) with interface, error handling and validating on Haskell.

The architecture of your solution.

Basically we can split this into two big parts: UI and Calculation, since terminal UI is too unusable for games like Sudoku, I actually developed a full-stack application. “Frontend” part is just a one single html file, and backend (Calculation) is a Scotty haskell application.

Why certain architecture decisions were done.

Why I choose:

to make a full-stack application: for developing more “serious” solution and for usability, it’s way more comfortable to use it in browser, rather than in terminal.

to use raw text in requests instead of JSON?: because the nature of my data (9x9 grid) is more suitable for just raw data, instead of JSON. I do not have a lot of structure, fields, different objects I have to pass. It’s

1. faster to send, receive, create and parse
2. easier
3. uses less memory and network

to use MRV heuristic, and not something else? because it’s not so hard (but it was challenging) to implement, and yet it gave a real boost to performance. We will talk about it later.

Why certain libraries were chosen.

The main decision and concern was choosing the right HTTP client. Since I didn’t have prior experience with backend development in Haskell, I needed something straightforward. So, I did my own research.

I chose Scotty, with Yesod and Servant being the other options I considered. Scotty is simpler and easier to use compared to Yesod and Servant. Yesod is a full-featured framework but overly complex for small projects, while Servant provides extreme type safety but requires advanced Haskell knowledge.

In terms of performance, the differences are minimal, as all of them are based on Warp, which, as far as I know, is the fastest server implementation in Haskell. It also took me some time to understand how Data.Text and Data.Text.Lazy work, but I don’t regret the effort in the end.

Investigation of the performance.

First of all, we have two functions here: the solver and the generator.

We can observe that any solution with some cells removed is still a valid Sudoku board. Similarly, any board that currently complies with the basic Sudoku rules (no duplicate numbers in a row, column, or box) is solvable. Therefore, we can generate a random solved board and then remove some values to create a puzzle.

The key task now is to understand how to solve Sudoku boards efficiently, which essentially boils down to optimized backtracking.

My solver implements the MRV (Minimum Remaining Values) heuristic, prioritizing cells with the fewest possible values first. This helps narrow down the search early. It also quickly rejects grids with invalid or out-of-range entries, saving time by avoiding unnecessary checks. By identifying cells with no valid moves upfront, it prunes unproductive paths immediately. These optimizations significantly reduce the amount of exploration needed, making the solver both faster and more efficient.

Since the algorithm's asymptotic complexity (O(N)) leaves little room for improvement, I focused on non-asymptotic optimizations and heuristic enhancements to improve performance.

My thoughts about the project.

I really enjoyed it. I have quite a bit of experience writing backends in languages like Golang, Kotlin, Python, and C++. Writing one in Haskell wasn't as hard as I thought it would be. That said, I did run into several issues with arrays and parsing text/grids, but I still thoroughly enjoyed it because it was something completely new and not just a fleeting experience.