Programming Paradigms in Practice 2023

Patrik Larsen 1800708

May 30, 2023

Multisweeper

Description

Multisweeper is a web application composed of a SvelteKit frontend and an Elixir backend. Since the assignment was exploring Elixir, I deliberately made the design choice of moving as much logic as possible to the backend. While this produced a more complicated architecture in general, it allowed me to delve deeper into the language in question.

How the features of the language and paradigm were used for implementing the game

One feature of Elixir I heavily utilized while creating the game was processes. Multisweeper creates a new process for each separate room created, as well as separates all requests into separate processes by utilizing the plug library. To keep track of room states and ensure synchronous creation/reading of rooms, they are managed by a GenServer. Processes are managed by a supervisor process, which ensures that critical processes are running and are restarted in the event of an unexpected failure.

The first thing I noticed when beginning to write Elixir was the functional nature of it, especially the immutability of data structures. Since the code base was quite small, I didn't really notice the benefits of immutability. Instead, I felt like it made many things more inefficient. The place where this immutability became the biggest problem was when I was implementing the search algorithm for finding connected empty cells, and I wanted to keep track of already explored cells. The solution I came up with was having an "exploration process" that had the sole responsibility of keeping track of explored cells. Immutability naturally resulted in me copying a lot of variables, which initially felt wrong, but which is apparently okay to do thanks to Elixir's aggressive garbage collector.

I also used atoms quite a bit and generally used tuples and lists a lot. Especially the Enum module was used extensively to iterate throughout the cells. When working with enumerable data, I also really liked the pipe operator, which made it easy to transform data through many functions effortlessly.

The match operator, while being extremely useful, wasn't really used widely, except in some cases where the program checks the result of some operations. Pattern matching in general wasn't used a lot either, except for fetching the head/tail of lists using the '[head—tail]' pattern, which I found very useful, especially in recursive functions when building lists.

Elixir's "Let it fail" mentality was embraced when building the application. If errors happen, we simply let the process fail and trust the supervisor to start up a new process when needed. Thanks to processes being separated, a room or a request crashing does not result in the application becoming unstable. As a result, the project doesn't use any try/catch blocks.

Analyze whether the language supported the development process or actually hindered the work

Since the project was quite simple, I felt like I didn't really experience the clarity of functional programming. After having worked on projects that aggressively mutate variables, I do, however, understand the value proposition.

The biggest way I felt the language hindered my work was by it being dynamically typed. Coming from having worked with TypeScript for a long time, it felt very irritating to not have the IDE check the types of variables constantly. If I were to do the assignment again, I would probably choose to use a statically type-checked language instead.