Functional Programming
Project: Chess (part 2)

Assistant: Noah Van Es
noah.van.es@vub.be

The final grade of the course is determined by an oral exam and a project (each counts for 50% of the total grade for the course). The project consists out of two parts. A non-empty solution for both parts has to be submitted in order to get a grade. This document describes the second part of the project.

Project description

For the second part of this project, you will have to apply four improvements to the terminal-based chess game you developed previously. Specifically, these involve:

GUI making a graphical user interface for the game using the Gloss graphical library.

Replay making it possible for the player to undo their last previous move(s).

Parallellism speeding up the AI algorithm by leveraging parallelism.

Persistency making the game persistent by adding save/load functionalities based on parser combinators.

For simplicity, your application should only support playing against an AI opponent.

Practical information

Submission The deadline for this assignment is **June 12, 2024 (23:59)**. You submit all your code and other materials in single ZIP-file through the Canvas platform.

Grading Note that your project will not only be graded on functionality, but also on the quality of your implementation (programming style, performance considerations, ...).

Individual work In case you have questions or need clarification to complete this project, feel free to contact the assistent by mail (`noah.van.es@vub.be`) or make an appointment to schedule a meeting (virtual or at my office). **This project is to be made strictly individually and on your own. This means that you must create your project on an independent basis and you must be able to explain your work, reimplement it under supervision, and defend your solution. Copying work (code, text, etc.) from or sharing it with third parties (e.g., fellow students, websites, GitHub, etc.) is not allowed. Electronic tools will be used to compare all submissions with online resources and with each other, even across academic years. Any suspicion of plagiarism will be reported to the Dean of Faculty without delay. Both user and provider of such code will be reported and will be dealt with according to the plagiarism rules of the examination regulations (cf. OER, Article 118). The dean may decide on (a combination of) the disciplinary sanctions (cf. OER, Article 118§5).**

Project assignment

This section details the specific requirements for each improvement.

GUI

The first improvement consists in creating a graphical user interface that shows the current state of the chessboard, and allows the player to make moves by clicking pieces on the board. As for the first project, the interface should prevent the player to perform any illegal move. In addition, **once the player has selected a piece on the board, all possible moves for that piece should be highlighted**. The player should be notified whenever the last move results in a checkmate or a stalemate.[1] As for the first assignment, promoted pawns can automatically be replaced by a queen. Finally, upon starting a new game, the player must choose via the GUI between playing white or black.

The GUI should be developed using the Gloss graphical library[2]. The exact rendering of the game is left to your discretion, as long as it supports all the functionalities described in this and the previous part of the project assignment. You can install Gloss using the following commands:

```
cabal update && cabal install --lib gloss
```

You should then be able to compile and run the following program:

```
import Graphics.Gloss
myWindow = InWindow "My Window" (200, 200) (10, 10)
main = display myWindow white (Circle 80)
```

To help you get started, you can have a look at some examples from the `gloss-examples` library[3]. These can be installed using Cabal and then executed as follows:

[1] For simplicity, it is not required to detect draw by threefold repetition or insufficient material.

[2] `https://hackage.haskell.org/package/gloss`

[3] `https://hackage.haskell.org/package/gloss-examples-1.13.0.4/src/`

```
cabal install gloss-examples
~/.cabal/bin/gloss-draw   # might be installation-dependent
```

The `gloss-draw` example is interesting to examine how events can be handled. Similarly, `gloss-conway` and `gloss-gravity` demonstrate how matrices can be rendered and how full-screen mode can be enabled, respectively.

Replay

The second improvement consists in giving the player the ability to undo their last move, in case they want to correct a mistake (and/or simply cheat to learn the game). Specifically, **the game should allow the player to go back any arbitrary number of moves**. This functionality should be integrated by adding an undo button to your GUI. Obviously, such an undo will also revert the last move made by the AI opponent.

Parallellism

Even the simplest board configurations can result in a large set of possible sequences of moves due to combinatorial explosion. As a result, the AI algorithm (cf. previous assignment), which has to enumerate all these sequences up to a given depth, can inherently become slow, resulting in poor scalability. **It is required that you attempt to mitigate the time spend by the AI opponent to compute the next move by leveraging the parallelism capabilities of Haskell**. The exact parallelism mechanism is left open to your own discretion, but you will have to argue for your design decision during the project defense. In order to execute your program on multiple processors, your code should be compiled with the `-threaded` flag along with the `-N` runtime system option.[4] Ideally, the time spent by the AI opponent for deciding the next move should decrease linearly with the number of processors allocated to the program.

Persistency

The final improvement consists in making the game persistent. That is, it should be possible to save the current state of the game to a file, and likewise load such a file to resume a saved game. **Your application should accept a single command-line argument, indicating the path to the file that should be used to save and load the game.** Concretely, it should support the following functionality:

- In case the file is not yet present, a new game should be started. Otherwise, the game should be loaded from the given file.
- A button should be added to the GUI to save the current state of the game to this file (either creating a new file or overwriting the previous one).

The format that should be followed for this text file is defined in Table 1. The current game state, as well as the game's parameters, are encoded in this format. Note that the game can only be saved on the turn of the player. If extra information is required, you are allowed to either (a) check/replay the history of moves or (b) add extra information to the serialisation format (in this case, you should document this change in comments).

[4] `https://wiki.haskell.org/Haskell_for_multicores`

Grammar Rule			Comments
GAME	→	PLAYER HISTORY PIECES	The game's serialization
PLAYER	→	player (COLOR) \n	The color chosen by the player
HISTORY	→	history (MOVES) \n	The history of previous moves
		ϵ	Omitted if the game just started
MOVES	→	MOVE	
		MOVE , MOVES	
MOVE	→	TYPE SQUARE SQUARE	Move without capture
		TYPE SQUARE SQUARE [TYPE]	Move with capture
PIECES	→	PIECE	Pieces on the chessboard
		PIECE \n PIECES	
PIECE	→	piece (TYPE, COLOR, SQUARE)	
COLOR	→	B	Black color
		W	White color
TYPE	→	king	Type of chess piece
		queen	
		bishop	
		knight	
		rook	
		pawn	
SQUARE	→	FILE RANK	A position on the chessboard
FILE	→	[a-h]	
RANK	→	[1-8]	

Table 1: The `.chess` serialization format

.