C(hes)S51 Report

Victor Alvarez & Nick Brinkmann

5 May 2021

1 Introduction

For our final project, we decided to implement chess in OCaml. This had been a longstanding goal of ours; we attempted a similar project for our CS50 final project in 2019, before realizing that we did not have the capabilities nor knowledge to successfully complete the project. This time around, we were much more successful, and implemented chess in its entirety, besides for the lone rule of threefold repetition, by which a draw is declared if the same position occurs three times during a game.
This paper will cover the different challenges, design decisions, testing and aspirations of our project. We will discuss how we went about implementing the different rules of the game, how we kept track of the state of the board, the way in which we rendered the graphic window and any challenges we encountered along the way. Afterwards, we will discuss other features that we would have liked to add if we had more time.

2 Design

We opted for an object-oriented approach to the game, whereby we implemented the chess pieces as subclasses of a common "piece" superclass. This piece-centric approach, rather than a square-centric approach, was convenient because it provided a divide-and-conquer approach with regards to the logic pertaining to the movement of pieces. This object-oriented approach was also attractive because the state of the board could be defined by a registry of piece objects that we handled and updated, meaning that we could freely create new objects and do whatever we wanted to them without changing the board state, as long as we made sure that our registry was unchanged after such manipulations. Having complete control over the registry was crucial in implementing many aspects of the project, as move validation and checking whether a player was in check, checkmate, or stalemate all required checking *every pseudo-legal move* for a player. This was because we needed to determine whether the player was inadvertently moving themselves into check, or had no pseudo-legal moves that would move them out of check.

Our implementation also relies heavily on a synthesis between the abstract logic of the game, and user-given input determined by the user's clicks in the graphics window. The user's clicks, for example, determine when a player has made his or her move. This is largely because, in our previous attempt, when we had alternated turns based on when a piece moved on the board, we ran into issues with castling, where the movement of the king and the movement of the rook were considered two moves, rather than one, as they should be. Logic related to the user's input in the graphics window also controls when the program evaluates whether a player is in check, as we had run into an infinite loop when this logic was handled purely in the objects and the registry module.

2.1 Infrastructure

When we first started the project, we had no clear idea of how to keep track of pieces, the game board, whose turn it was, etc.. We experimented with the following definitions:

```
type file = A | B | C | D | E | F | G | H ;;
type rank = R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 ;;
type piece = Pawn | Knight | Bishop | Rook | Queen | King ;;
type coordinate = file * rank ;;
type square = piece option * coordinate ;;
```

From this we were considering having an 8×8 array where each entry would be of type "square" and we'd simply manipulate the entries in this array. However, we quickly realized that it would be useful, and in some cases necessary, to keep track of data such as how many times an individual piece has moved, what defines its valid moves and whether or not a player was in check.
At this point, rather than create several global functions, we decided to create classes for each piece where we would define methods for move validation, drawing themselves in the graphics window, and keeping track of their own position after each move. Then, we took inspiration from problem set 8 and decided to use a registry (implemented as a set) to keep track of which pieces were currently on the board. Taking advantage of this registry module, we defined a lot of the game logic as functions within this module.
We initially had an 8×8 array within the registry defined as

```
let position = Array.make_matrix 8 8 None
```

which was only used for quickly finding a piece at a specific position, since we could simply index into it. However, seeing as this was its only purpose we decided to remove it entirely and instead search for pieces by filtering the registry with a boolean function. In hindsight, an array may actually proved useful in checking for threefold repetition (when the same board position occurs three times in a single game) and for extra safety in preserving invariants such as never having more than one piece on any particular square.

2.2 Game Logic

The logic of the game can be explained as the following algorithm:

1. Draw the board, the pieces at their respective positions, and any necessary alerts to the player whose turn it is.
2. Wait for the player to select a piece to move (the player can change their selected piece as many times as they want, but a piece must be selected before moving on to next step)
3. Wait for the player to select a square to which they will move the selected piece
4. If specified move abides to the way the piece can legally move then move on, otherwise go back to step 3.
5. Make copy of the state of the game before the move (making a copy of registry of pieces)
6. Execute move provisionally, thereby updating the current registry
7. If move would leave player in check then revert back to previous state of the game (re-register pieces using copy taken in step 5) and return to step 2. Otherwise, continue.
8. Check the state of the game for next player after this move is taken
9. Repeat steps 1 - 8 until checkmate or stalemate is achieved

The logic of the game is not centered on any single file, but rather depends on the interaction between functions defined in the piece classes, the registry and the visualization file. The visualization file primarily serves to provide the correct inputs to the methods and functions defined in the registry and the class methods. The visualization file draws all of the elements in the graphics window: the board is drawn in the same way each time, alerts are drawn depending on the state of the game (a variable set in the registry module), and the pieces are drawn by iterating through the registry of pieces and calling their draw methods. Input is taken by user clicks inside the graphics window. After each move, the program blocks waiting for clicks inside the window. It then uses the x and y coordinates of the mouse at the moment the click occurred, and decides what to do based on those inputs. By converting these coordinates into chess coordinates (a file and a rank) the program decided whether to select a piece, move a piece, or take back a move. A turn was dictated by two actions, selecting a piece and then selecting a square to move it to. Alternatively, a player could click the 'take back' button and the game would revert to the previous position. Most of the game information was dealt with in the registry.ml file. Within this registry module we had functions to find pieces, take back a turn, check the state of the game along with other functions that were used in move validation. Once a click on the board was registered in the visualization file it called on the

registry module to find if a piece was at the square that was clicked on so that it could be selected. If a piece has been selected the game waits for the user to click on another square on the board to attempt to move the piece there. Note that if the following click was on one the player's own pieces, it would simply swap the 'selected' piece.
The pieces.ml file contains the definition of each piece class, along with their 'can_be_valid_move' and 'make_move' methods. If in visualization a piece was found, selected, and a move was made, it would call the 'can_be_valid_move' function. This function simply checks that the square to which the piece is attempting to move is within the legal definition of how this piece moves. For rooks this would be checking that it is along a horizontal or vertical line form its starting position, with no pieces in the way. Similarly for bishops, but along diagonal lines.
If the attempted move was deemed pseudo-legal, we would call the 'take_turn' function defined in the registry. This function takes a copy of the registry as it is in that moment, before the move was made. Then the move was executed provisionally by calling the piece's 'make_move' method. This method updates the pieces position and, if the move involved taking a piece, would deregister the necessary piece. After doing so we called the 'player_not_in_check' function defined in the registry. This function would verify that after the move was made, the player would not be left in check. Otherwise the move would not be legal as you cannot move yourself into check. If the function returned that the move would leave the player in check, then a call was made to the 'take_back' function, reverting the state of the game to its original position and alerting the player that this would not be a valid move. Otherwise the move was considered complete and the registry would have already been updated. Before the next player takes their turn, we call on 'checkmate_check', 'stalemate_check' and 'player_not_in_check' to determine whether the move that was just played has put them into checkmate, stalemate, or check. We then update the state of the game accordingly.

3 Challenges

We experienced significant challenges and setbacks in our implementation.

- Relatively early on in the project, we experienced a bug where, after a piece had captured an opponent's piece, it would raise an exception if it were itself captured. After lengthy sleuthing and the invaluable help of Jackson, our TF, we discovered that this was due to a faulty comparison function of our set of pieces, where two pieces were considered 'equal' if they occupied the same square. The issue was resolved by determining a more suitable comparison function.

- We ran into an infinite loop, which occurred when we tried to validate whether a piece could move to a given square. The piece classes each had their own method, called 'can_be_valid_move', which determined whether

a piece could legally move to a given square. To check this, we attempted to implement a function in our Registry module that determined whether a player of a given color was in check. The idea was that if a move didn't put a player in check, and it was a pseudo-legal move, then it could be played. The issue that arose was that, to check if a player is in check, you need to check if it is being attacked by enemy pieces: that is, *if it is a valid move for those pieces to move onto the king's square.* To evaluate that, we would have to call can_be_valid_move on all of the opponent's pieces, which would cause an infinite loop and eventual stack overflow. This is actually quite a subtle and challenging issue. To solve this, we instead moved the call of the 'is_player_in_check' function outside of the 'can_be_valid_move' method and into the realm of the graphics logic determined by user clicks. This way, after making a provisional move, we could check if the opponent's pieces were attacking the king without having to call the 'is_player_in_check' function for the opponent.

- The logic of special rules was quite difficult. For castling, our implementation needed to keep track of whether a player's king and rooks had ever moved before (which necessitated a move counter for each piece), whether the player was currently in check, whether there were any pieces between the king and the rook, and whether the opponent's pieces were attacking the king's ending square or the intermediate square (as a player cannot castle through check). Besides this complicated logic, as mentioned in the introduction, we ran into an issue where castling was counted as two moves for a time (one for the king, and one for the rook), which threw our tracking of the game state into disarray as, after a player had castled, the game thought it was their turn to move again, because 2 turns had passed since the player's last move. This also made taking back moves complicated, as the user would have to take back twice to get to the position before castling. As mentioned in the introduction, we solved this by having our turns tracked not by the move method of pieces, but by the user's clicks in the graphics window.

- En passant, meanwhile, necessitated keeping track of the previous move that had been played. Ultimately, since we also had a take back feature, we were required to keep track of all previous moves as well as the corresponding registries (positions). We also ran into a more conceptual issue in en passant. Due to some complicated logic, we had to call 'take_turn', a function that kept track of the move history and all moves, *before* actually moving the piece. This was because, for example, if a player's knight was pinned to its king, and we moved it and took an opponent's piece, if we had moved the piece before calling take_turn, that would have deregistered the opponent's piece, thus making it difficult for us to restore the position prior to the illegal move. However, this meant that en passant was buggy in cases. Because we update the move history *before* moving a piece, when we execute en passant, we need to check not for the *last* move in the move history (which now would have been the en passant

move itself!), but rather, the *penultimate* move in the move history. This is not the slickest solution, but again, an alternative solution that would not have raised other bugs or complications in other aspects of the game is not obvious.

- We needed to use copies of piece objects, rather than the pieces themselves, when performing move validation and checking whether a player is in check/whether they have any valid moves. This was largely because of the requirement that to castle, a king and its rooks cannot have moved yet. If we had used the real objects in these checks, it would have incremented their move counters, thus rendering castling (and, in an earlier implementation, en passant) invalid. Using these copies of pieces also introduced difficulties since we had a persistent bug that would not detect checkmate, because in checking for checkmate, the function that checked for checkmate used the current registry, while the function that checked whether a piece had a valid move was iterating through the pieces in the registry and calling the function that evaluated whether it was check, and by the time this iteration reached later pieces, the registry had been altered in the process of checking whether the prior pieces had valid moves. (If this explanation seems hard to follow, imagine what we went through while trying to debug it!) Besides this conceptual difficulty, this approach also means that our checks are currently rather inefficient, as we need to update the registry after every checked move. However, an alternative solution that does not introduce potential errors with move validation (particularly castling, en passant, and checking for check/checkmate/stalemate) is not obviously available.

- When a pawn is promoting, we set up a graphic that gives the user the option of what to promote the pawn to. We ran into an issue when checking valid moves, wherein if a pawn was on the seventh (or second, for black) rank, this graphic would come up automatically without the user actually attempting to promote the pawn, due to the program *checking* whether the pawn could promote. We grappled with this for a while, ultimately realizing that our program would need to distinguish between promotions made by the user, and promotions made in performing validation checks. While it is not an ideal solution, we settled on the use of a global boolean variable 'is_real_board', such that the promotion portion of the pawn's move method is only entered if this variable is true. An alternative solution would have been to reconfigure all of our code to take an optional argument indicating whether the user had made the move, or whether the move was being performed for validation checks. While this was potentially a better option, it was not feasible given our limited time constraints.

4 Testing & Invariants

Almost all of our testing was done by hand, given that much of our project relies on the integration of the abstract logic of the game and user-given input in the graphics window. In our presentation video, we verify that many of the important cases (en passant, castling, castling through/out of/into check, moving into check, promotion/underpromotion, checkmate/stalemate) behave as they should. In addition, we made use of several checks and invariants to make sure that our code was behaving as desired. For example, we made sure that pieces could never move onto a square that was occupied by a friendly piece; that in en passant, there was always an opponent pawn directly above/below the taking piece; that we never deregister a piece that is not already in the registry; that, in find_piece, there is only ever one piece on a square, and other checks and balances. In self-testing our implementation, we did our best to hunt for edge cases, but of course, it is possible that there are some cases that we did not consider.

5 Further Improvements

Although we are extremely proud of how the project turned out, there are definitely areas that we could improve and features that we would have liked to add. We will add brief descriptions of what we think we might have done for some of these.

- **Three fold repetition**: we might have considered saving the state of the board as an array after each move, and adding it to a dictionary whose key/value pairs are an array, and how many times we've seen it. Then, when we make a move we check if we've seen it before and increment its counter, otherwise we add a new entry with the counter starting at 1. If at any point a counter reaches 3 then we've seen the same position 3 times and we end the game as a draw.

- **Printing moves to the side of the board**: In debugging we already had ways of printing out moves in the terminal. We could have simply moved the same logic over to the graphics window so that we could print out the moves. The only concern would be running out of space in the graphics window.

- **Lighting up valid moves once you select a piece**: In checking for checkmate and stalemate we already have a function that searched for any valid move. We could simply extend this function to check for *all* valid moves given a piece and light up the squares that it returns.

- **Adding more buttons (new game, resign, etc.)**: We would basically use the exact same logic as our current buttons and add helper functions that would be called when a button is clicked. These helper functions

would do things like reset the registry to the starting position or end the game entirely.

- **Printing actual pieces rather than their names**: We attempted to use Camlimages to convert png images of chess pieces into OCaml graphics module-friendly images, but we ran into difficulties setting this up on our systems. Ultimately, we were not able to do this, despite the much-appreciated help of Jackson and Ahan. If we had more time, we likely could have enlisted further help and figured it out. Seeing that this was a purely aesthetic improvement, it was relatively low on our priority list.

- **Separate game logic and graphics logic a little more**: In our current implementation, too much of the game's logic is exposed to the user. The Registry module, for example, exposes too many functions that leave our code open to malicious abuse. If we had more time, it would be nice to think more carefully about the functionality this module exposes, and to try to cut down on unnecessary functionality to prevent such abuse.

- **Adding ability for command line arguments/keyboard input**: In the early days of the project we experimented with the graphics module, which has functions for polling/blocking for keyboard input. We could have taken keyboard input that way, and converted the characters into commands that could be executed. If we had more time perhaps we may have implemented this, but the project was largely implemented based on clicks and mouse coordinates and we did not have the time to refactor all of the logic to account for keyboard input as well.