# Haskell Solitaire Assignment Report

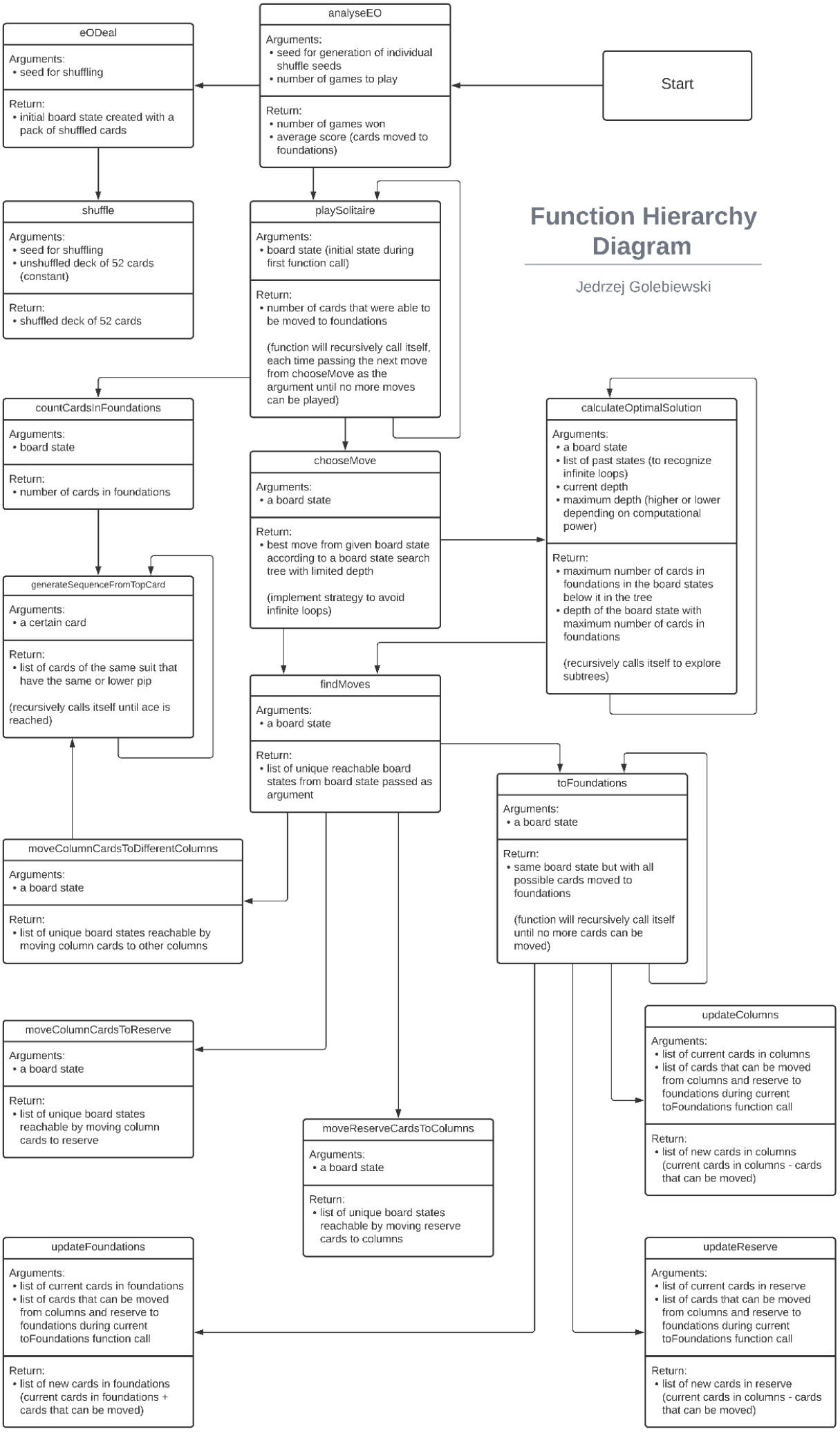
## Design for eight-off solitaire

### Data Structure Definitions

1. **Suit** - One of the 4 suits in a normal deck of cards.
2. **Pip** - One of the 13 different values in one suit.
3. **Card** - Tuple containing one **Pip** and one **Suit**.
4. **Deck** - List of **Card**.
5. **Foundations** - List of **Card**.
6. **Columns** - 2-d list of **Card**.
7. **Reserve** - List of **Card**.
8. **Board** - Can be either an EOBoard (eight off solitaire) or SBoard (spider solitaire).
   EOBoard has 3 parameters - **Foundations Columns Reserve**
   SBoard has 4 parameters - **Foundations Columns Hidden Stock**
9. **Hidden** - List of (**Int**, **Int**) tuples, where the first **Int** is the index of the column in **Columns** where some cards are to be hidden and the second **Int** is the index in the column, where all the cards with indexes equal or higher are supposed to be hidden.
10. **Stock** - List of **Card**.

### Function Specifications and Hierarchy

(on next page)




Function Hierarchy Diagram for Eight-Off Solitaire

# Experimental results for eight-off solitaire

## Strategy

The strategy that I used in determining the best move from all the possible moves from a given board state was an iterative deepening depth-first search with an initial depth limit of 1 and increment of 1 to the depth each time the search is not able to find a board state with more cards in foundations compared to the initial board state at the root. Once a board has been found with more cards in foundations relative to the root, the first move on the path to that board state is chosen. If multiple boards that increase cards in foundations are found at a given depth, the one that has the largest number of cards in foundations is chosen.

Even though the chooseMove function has no way of tracking past board history due to the function definition specified in the assignment brief and its use in template.hs (which means it can't have a wrapper function around it with board state tracking), the function will never enter infinite loops and will converge on a solution or terminate due to the nature of the search strategy.

The chooseMove helper function (calculateOptimalSolution) however does keep track of all the past board states on a path from the root to each node and if a node being evaluated encounters a previously discovered board state on the path, then further exploration from the node is terminated. Although this is not necessary, because the search continues to go on until it encounters a move which increases cards in foundations (and these are one-way moves which can't cause infinite loops), it is an optimization strategy.

Another optimization strategy is not forcing a search to the maximum depth limit each time the chooseMove function is called and taking the path which has the largest foundations card no. increase, but instead stopping at the first encounter of a foundations card no. increase (if the increase happens in multiple boards at a depth level, the one with the largest increase is chosen). Although this is a greedy strategy that optimises locally, it seems that stopping the search on the depth level of the first available move that moves cards to foundations, no matter how many, has no major drawbacks.

## Results

| **Max Depth** | **analyseEO seed: 12, no. of games: 200** | **analyseEO seed: 25, no. of games: 200** |
|---|---|---|
| 1 | 0 wins, 3.77 avg. cards in foundations | 0 wins, 3.39 avg. cards in foundations |
| 2 | 4 wins, 8.52 avg. cards in foundations | 3 wins, 7.2 avg. cards in foundations |
| 3 | 26 wins, 15.07 avg. cards in foundations | 25 wins, 14.375 avg. cards in foundations |
| 4 | 59 wins, 22.0 avg. cards in foundations | 62 wins, 22.675 avg. cards in foundations |
| 5 | 87 wins, 27.71 avg. cards in foundations | 84 wins, 26.765 avg. cards in foundations |

## Summary of results and advantages/disadvantages of strategy

It is clearly noticeable that there is a positive correlation between the maximum depth permissible in the search and the results. The strongest improvement in results seems to be between maximum depth 3 and 4. I believe that due to the number of games played (200), any individual variances in the initial board states can be ignored and the law of large numbers can be applied. Unfortunately, it is quite hard to do test runs past max depth 5 because of the time required to run such a test.

I think the strategy could be optimised by the use of a combination of heuristics to guide the search, e.g. the ones mentioned in the assignment brief. Currently the strategy is a strictly brute force one and this is noticeable in the execution time of the program. The advantage of this strategy however is that, given enough computational resources and removing the max depth limit, it should be able to solve most solvable solitaire games.