Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC Sr Eberl

WS 2020/21
18.12.2020
Deadline: 11.01.2021, 23:59

Functional Programming and Verification

Sheet 8

Homework

You need to collect 8 out of 10 Christmas stars ($\star$) to collect a coin.

Exercise H8.1 So This Is Christmas Volume 2: *Virus vs Antibodies* [a–d: 1$\star$ each, e: 2$\star$, f: 4$\star$]

The MCs are drinking mulled wine, it is snowing on the lecture website, in other words: it is Christmas time. And as it is tradition, every Christmas, the MC Jr organises a board game night with his family, this year being no exception. While last year all parties involved had to assemble domino blocks in clever ways, this year's game, for no obvious reason, is a two player game in which a virus fights against antibodies.

You must know, the MC Jr hates losing the Christmas board game night. Thanks to last year's FPV-students, however, his family stood no chance and the MC Jr was victorious last year. To repeat his success, he wants to make sure that he gets enough practice before playing against his family this Christmas. Sadly, real social interactions are an impossiblity these days and hence practicing with his friends is out of the game; however, the MC Jr has another brilliant idea: he simply instructs you to write an AI with which he can practice.

The main idea of the game is to simulate the fight between viruses and antibodies. Below, you can find the rules of the game. Note that this is a functional programming lecture and not a virology course, so there may be some medical inaccuracies – forgive us Mrs Ciesek and Mr Drosten:

- The game is played by two players: Player +1 (Virus) and Player -1 (Antibodies).
- The game takes place on an $n \times m$ cell grid, where $n, m \geq 2$.
- Initially, each cell is empty.
- In each turn, a player can either claim an empty cell by placing an orb in it or place another orb in a cell that player already owns.
- A player cannot place orbs in cells claimed by the opponent.
- A cell is *filled* when it contains at least the same number of orbs as orthogonally adjacent cells.
- When a cell is filled, the virus/antibodies overflow and attack the orthogonally adjacent cells, moving one orb into each neighbour and converting existing orbs into their own kind.
- The game ends when either viruses or antibodies reach their goal of eliminating all orbs of the enemy.

A game board can be displayed by a two dimensional list, where each entry is either 0 if it is unclaimed, $+n$ if player +1 owns this cell and has n orbs in it, or $-n$ if player -1 owns this cell and has n orbs in it.

For example, if player +1 puts an orb at position $(0, 1)$, the board `[[1,2,-1],[2,0,0],[0,-1,1]` transitions to `[[1,2,0],[0,2,1],[1,-1,1]]`. Note that there were overflows in all cells of the top row and in cell $(1, 0)$.

Your task is to implement a strategy that chooses the next cell to place an orb in. To do so, the MC Jr's diligent tutors started to implement a framework for simulating such games. However, just like last year, the MCs and all tutors are really busy buying Christmas presents *auf den letzten Drücker* and were not able to finish the framework. You hence need to implement a few more utility functions before you can write your strategy.

In the following, you may assume that all inputs are valid arguments – unless, of course, you yourself pass potentially invalid inputs to some function. Here are the missing functions:

a) `canPlaceOrb :: Player -> Pos -> Board -> Bool` that checks if a player can put an orb at the specified position on the given board.

b) The function `hasWon :: Player -> Board -> Bool` checks if the board state is in a winning state. A game is won if the opponent controls no cells. It is additionally provided with the player that made the last move (the only player that could have won). You can assume that all players have made at least one move.

c) `neighbours :: Size -> Pos -> [Pos]` takes a position on a board of a given size and returns a list of neighbouring positions. The order of the resulting list does not matter. Example: `neighbours (4, 6) (3, 5) = [(2, 5), (3, 4)] = [(3, 4), (2, 5)]`

d) Next, implement `updatePos :: (Int -> Int) -> Player -> Pos -> Board -> Board`. This function modifies a single cell on the board by changing the number of orbs in the cell with the provided function and assigns the result to the passed player. You do not need to handle any overflows here.

e) Finally, the function `putOrb :: Player -> Pos -> Board -> Board` should place an orb for the given player at the given position. If this causes the cell to be filled, the cell should overflow and attack the neighbouring cells as described above. This might cause a neighbouring cell to overflow which will then attack neighbouring cells and so on, until a stable state is reached. This process can continue after one player has won, potentially never stabilizing. Make sure that your solution detects this and stops in a valid, reachable state.

f) Now you can finally implement your `strategy`. You are provided an infinite[1] list of random numbers, your team (+1 or -1), and the current board state and you must return a position to put an orb onto. Make sure that your strategy takes no longer than 1 second of CPU time per move; for otherwise, you shall loose the game instantly.[2]

[1] We will talk about infinite lists later in the course; as for now, you can just assume that the passed list will always contain one more element if needed.

[2] If your move takes less than 1 second, the remaining time will actually be added as a credit to your time balance

If your strategy needs to preserve some state between calls, you can implement `strategyState` instead. This is a tuple of an initial state and a strategy function that additionally receives the state and also additionally returns the next state. You may change the type argument `Int` of `StatefulStrategy` to any other type that is useful for you. Your change is valid if the call `playAndPrint defaultSize strategyState strategyState` still works in ghci. Do not modify any other given type annotations. Do not remove `strategyState` even if you do not make use of it directly.

You can simulate games with
`playAndPrint defaultSize (wrapStrategy strategy) (wrapStrategy strategy)` or `playAndPrint defaultSize strategyState strategyState`. The output is the board state that each call to a strategy receives, together with the selected position. Additionally it will generate a link where you can view the entire game in your browser – pretty cool, no?

For the homework, you will face off the virtualisation of the MC Jr's grandpa and sister in four rounds. You will be awarded one glamorous ⋆ for each win.

Wettbewerb: The MC Jr seeks for challengers, no victims. He badly needs to win the upcoming familiy Christmas board game night. His sister is a mastermind, but she shall stand no chance if he practices using only the very best strategies as submitted by you. Help him being triumphant yet another time and his gratitude will be tremendous.

Implement the best strategy challenging the MC Jr, best being defined as the strategy that wins the most games. To help you gauge the quality of your submission, the MC Jr's tutors went absolutely crazy and set up a fully-fledged competition website including rankings, statistics, playbacks, and even bought a fancy domain: https://virusga.me.

Your submission will be simulated against your fellow students' submissions as soon as possible given that you were able to beat the MC Jr's grandpa on Artemis and you included the `{-WETT-}...{-TTEW-}` tags. Please be patient though – there are quite a few students registered for FPV. . . **flashbacks about last year's simulation bugs and out-of-memory problems**

This exercise was designed and implemented in coorporation with our tutors. Special thanks to all of them!

> Haskell is faster than C++, more concise than Perl, more regular than Python, more flexible than Ruby, more typeful than C#, more robust than Java, and has absolutely nothing in common with PHP.
>
> — Audrey Tang

for future invocations. However, you do not really have any way to measure the time during your strategy invocation, so this might be more of a gimmick rather than being useful.