AFP - Takuzu - Report

Maximo Garcia Martinez & Koen Timmermans

1. Structure of the project

This project has one file, `tic-tac-logic.hs`, which can be compiled using `ghc` and used as in the assignment. The solver is implemented using constraint programming, roughly following theory from the course Combinatorial Optimisation and Constraint Programming (1DL441)[1].

A `Grid` represents a (partially solved) puzzle and consists of a list of equally length lists of `Cell`s, which can be `X`, `O` of `U` (undefined). Each solving technique is implemented using a *propagator*, which takes a grid, applies the technique once on all rows and all columns, and returns the new grid and a message. The message can be `Failed`, in which case we know that the input grid was not solvable (and the result grid must be ignored), `AtFixpt`, which happens if nothing changed to the grid this propagation step, and `Unknown`, when propagation succeeded but a repeated application of the propagator may improve the grid.

The propagators are implemented using *maskers*, which take a grid and either `X` or `O` as its argument, and return a *mask*, which is a grid filled with `U`s, and either `X` or `O` in places where the propagator wants to add an `X` or an `O` to the existing grid. For example, the masker for basic technique 1 and `X` on the row `OO.XXO.O` returns `..X.....`, because that is the only `X` that should be added according tot only basic technique 1.

Each propagator creates the `X` and the `O` mask for its technique of the input grid, and generates a message and an output grid. The message is `Failed` if either both masks do not agree about a cell (using the example above, the masker for basic technique 1 and `O` would return `..O.....`, so the third cell has to be simultaneously `X` and `O`, which makes the input grid insolvable) or if applying the grid would result in an incorrect grid (with three equal marks next to each other or with double rows/columns). The message is `AtFixpt` and the original grid is returned if both masks only contain `U`, and the message is `Unknown` and the grid with applied masks is returned in other cases. Propagation is scheduled according to the algorithm on page 27 of the slides of topic 14[2] of the above mentioned course and is implemented in `propagate` and `propagaterec`.

Solving a grid happens by first propagating the grid and then, if the grid is not solved, guessing `X` for an empty cell and solving that grid. If the grid is solved, `Just grid` is returned. Solving can result in a failure, in which case `O` for that cell is tried. If that again fails, `Nothing` is returned. This happens in the functions `solve` and `search`. The cell that `guess` selects is the first cell in the row that has the least non-zero amount of unknowns. This way, propagation is expected to solve a larger part of the new grid than just selecting the first empty cell, which speeds up solving.

The puzzle in `tic-tac.txt` is solved by running either `cat tic-tac.txt | ./tic-tac-logic` in the command line or `mainfile "tic-tac.txt"` in the interactive compiler. If the puzzle cannot be solved or if the input format was wrong, `No solution` will be displayed.

2. How did we divide the work?

We have done the projects separately, but we checked each others work. This project has been made by Koen.

3. Extra requirements fulfilled

No testing functions are implemented.

[1] `http://user.it.uu.se/~pierref/courses/COCP/course.html`
[2] `http://user.it.uu.se/~pierref/courses/COCP/slides/T14-Propagation.pdf`

Advanced technique 2 is implemented. Basic technique 5 the special case of advanced technique 2 where only one `X` and one `O` are missing, so it is automatically implemented as well. Advanced technique 1 is not implemented.

4. Haskell's unique features

Higher order functions are heavily used. A lot of the functions on grids are maps of functions on rows, and other functions are dependent on functions passed to them as well. For example the function `propagate` maps an integer to a propagator, and a propagator itself is a function that converts grids into other grids. Also the function `makepropagator` depends on maskers which are functions themselves. This is streamlined by the use of type synonyms, the type of every function in the file is given, and the types `Propagator` and `Masker` are defined to abstract away from the fact that they are actually functions.

Monads are also used, mainly the `Maybe` monad and lists. Some functions, such as `basic3try`, `cttry` and `oneelt` are based on list comprehensions, which describe monadic operations on input lists. The `Maybe` monad is used in order to make the solving process recursive. After a guess is made, `solve` is ran on the new grid but that new grid might not be solvable. So `solve` returns an instance of `Maybe Grid`, which is `Just grid'` if `grid'` is a solution, and `Nothing` if no solution is found. It is combined with the function `inputCheck` that also returns a `Maybe Grid` using the bind operator. And `solve` itself is also defined using the bind operator, because it calls `propagate` first which may return `Nothing`.

Thirdly, laziness is used in for example the `correct` and `solved` functions. As soon as those functions find that the input grid is not correct/solved, they will return false. Haskell does this automatically, no extra logic has to be implemented for this.

5. Material that we have used

Theory from the above mentioned course is used to create the solver.

A few test puzzles are incuded in the file, they are called {`1--7`}`.txt`. The file `5.txt` is the 'larger example' in the project's description. The solver can solve it in 2.8 seconds on Koen's machine.

The file `puzgen.sh` is also included, which creates empty puzzles of a given width and height, this was used to see how well the solution scaled and how execution time changes using different propagators. The solver can solve empty puzzles until 16×14 within 10 seconds, but this is not representative for puzzles with an unique solution since a lot of (wrong) guessing has to be done before any propagation can happen and the complete puzzle can be filled in.