Lab 3 -- Extra Assignments

Just for fun. You can choose freely whether to do 0, 1 or more of these. Don't expect us to spend time grading these however. There are no perfect, pre-defined answers here.

X. Perhaps if we picked the blank in a smarter way, the **solve** function would go faster?

One idea is to always pick the blank spot where there are **as few possibilities left**. For example, if we have a row with one or two blank spots, it is probably a good idea to pick one of those blank spots, since it will limit the consecutive search most, and it will lead to search to a state with more digits filled in. (Such a way of changing a solving method is called a **heuristic** -- there is no absoluate guarantee that the search will go faster, but often it actually will.)

Does your **solve** function work faster now? Experiment with different heuristics (for example: only look at rows and columns, and not at 3x3 blocks), and see which one performs best. Can you solve some of the hard Sudokus now?

Do not forget to add appropriate properties that test your functions.

Y. The solving method we have used in this lab assignment is very basic, and in some sense naive. The best known methods to solve problems like Sudoku is to also include the notion of **propagation**. This is the way most humans actually solve a Sudoku.

A simple variant of propagation is the following. Suppose we have Sudoku with a row with precisely one blank, such as the 3rd row in the example below:

```
36..712..
.5....18.
..92.47..
596.13428
4..5.2..9
27.46....
..53.89..
.83....6.
..769..43
```

Our current solution would go and pick blanks, and start searching recursively, without making use of the fact that we already know the value of that blank (namely 7 in this case); all the other values have been used by the other cells in the row.

Implement a function

```
propagate :: Sudoku -> Sudoku
```

that, given a Sudoku, finds out which rows, columns, and 3x3 blocks only have one blank in them, and then fills those blanks with the only possibly remaining value. It repeats doing this until all rows, columns and 3x3 blocks are either completely filled up, or contain two holes or more.

Now, add this function at the appropriate place in your solve function. Does it work faster now?

For other, more powerful propagation, you can for example read the following webpage:

- **sudoku.com**, and click on "how to solve".

Or come up with your own propagation rules!

Do not forget to add appropriate properties that test your functions.

Z. Write a function that produces interesting Sudoku puzzles. For example, one could have a function

```
createSudoku :: IO ()
```

that every time we run it, would print a new, interesting Sudoku puzzle on the screen.

One can discuss what an interesting Sudoku puzzle is. Here are three properties that an interesting Sudoku puzzle must have:

- There must be a solution
- There must not be two different solutions
- There must not be too many digits already visible

Can you think of a way to define a function for generating an infinite supply of new Sudokus satisfying the above two properties? You should of course make use of the functions you already have.

Do not forget to add appropriate properties that test your functions.

Å. Generalize the Sudokus that are dealt with in this assignment to other dimensions, for example 4x4 Sudokus, or 4x3 Sudokus. Do all dimensions make sense? Make sure your solution works in general, for all possible dimensions that make sense.

For inspiration, look here: **Monster Sudokus**. Or here: **xkcd comics** :-)

Do not forget to add appropriate properties that test your functions.

Ö. We have stated the **soundness** of the **solve** function as a property; every produced solution should be a real solution. The dual of soundness is **completeness**. Completeness says that whenever there is a solution, the **solve** function should also produce a solution. (Equivalently, if the **solve** function says that there is no solution, then there really is no solution.)

If we define the following helper datatype:

```
data SolvableSudoku = Solvable Sudoku
```

Then, implement a property

```
prop_SolveComplete :: SolvableSudoku -> Bool
prop_SolveComplete (Solvable sud) = ...
```

that states that, for any solvale Sudoku, **solve** produces an answer.

Now, make the type SolvableSudoku an instance of Arbitrary:

```
instance Arbitrary SolvableSudoku where
  arbitrary = ...
```

Here, you need to think about how to generate arbitrary Sudokus that are guaranteed to be solvable!

One idea is to start with an arbitrary solved Sudoku, and randomly blank out some of the digits. Implement this!

Back to Lab 3 description