Fast Sudoku Solver in Haskell #1: A Simple Solution

June 28, 2018
A thirty-one minute read
Tags: haskell, sudoku, programming, puzzle, nilenso

Sudoku is a number placement puzzle. It consists of a 9x9 grid which is to be filled with digits from 1 to 9. Some of the cells of the grid come pre-filled and the player has to fill the rest.

Haskell is a purely functional programming language. It is a good choice to solve Sudoku given the problem's combinatorial nature. The aim of this series of posts is to write a **fast** Sudoku solver in Haskell. We'll focus on both implementing the solution and making it efficient, step-by-step, starting with a slow but simple solution in this post[1].

This is the first post in a series of posts:

1. Fast Sudoku Solver in Haskell #1: A Simple Solution
2. Fast Sudoku Solver in Haskell #2: A 200x Faster Solution
3. Fast Sudoku Solver in Haskell #3: Picking the Right Data Structures

Discuss this post on r/haskell.

Contents

Constraint Satisfaction Problem

Solving Sudoku is a constraint satisfaction problem. We are given a partially filled grid which we have to fill completely such that each of the following constraints are satisfied:

1. Each of the nine rows must have all the digits, from 1 to 9.
2. Each of the nine columns must have all the digits, from 1 to 9.
3. Each of the nine 3x3 sub-grids must have all the digits, from 1 to 9.

```
+-------+-------+-------+
| . . . | . . . | . 1 . |
| 4 . . | . . . | . . . |
| . 2 . | . . . | . . . |
+-------+-------+-------+
| . . . | . 5 . | 4 . 7 |
| . . 8 | . . . | 3 . . |
| . . 1 | . 9 . | . . . |
+-------+-------+-------+
| 3 . . | 4 . . | 2 . . |
| . 5 . | 1 . . | . . . |
| . . . | 8 . 6 | . . . |
+-------+-------+-------+
    A sample puzzle
```

```
+-------+-------+-------+
| 6 9 3 | 7 8 4 | 5 1 2 |
| 4 8 7 | 5 1 2 | 9 3 6 |
| 1 2 5 | 9 6 3 | 8 7 4 |
+-------+-------+-------+
| 9 3 2 | 6 5 1 | 4 8 7 |
| 5 6 8 | 2 4 7 | 3 9 1 |
| 7 4 1 | 3 9 8 | 6 2 5 |
+-------+-------+-------+
| 3 1 9 | 4 7 5 | 2 6 8 |
| 8 5 6 | 1 2 9 | 7 4 3 |
| 2 7 4 | 8 3 6 | 1 5 9 |
+-------+-------+-------+
    and its solution
```

Each cell in the grid is member of one row, one column and one sub-grid (called *block* in general). Digits in the pre-filled cells impose constraints on the rows, columns, and sub-grids they are part of. For example, if a cell contains `1` then no other cell in that cell's row, column or sub-grid can contain `1`. Given these constraints, we can devise a simple algorithm to solve Sudoku:

1. Each cell contains either a single digit or has a set of possible digits. For example, a grid showing the possibilities of all non-filled cells for the sample puzzle above:

```
+-------------------------------------+-------------------------------------+------------------------------
| [123456789] [123456789] [123456789] | [123456789] [123456789] [123456789] | [123456789] 1           [12345
| 4           [123456789] [123456789] | [123456789] [123456789] [123456789] | [123456789] [123456789] [12345
| [123456789] 2           [123456789] | [123456789] [123456789] [123456789] | [123456789] [123456789] [12345
+-------------------------------------+-------------------------------------+------------------------------
| [123456789] [123456789] [123456789] | [123456789] 5           [123456789] | 4           [123456789] 7
| [123456789] [123456789] 8           | [123456789] [123456789] [123456789] | 3           [123456789] [12345
| [123456789] [123456789] 1           | [123456789] 9           [123456789] | [123456789] [123456789] [12345
+-------------------------------------+-------------------------------------+------------------------------
| 3           [123456789] [123456789] | 4           [123456789] [123456789] | 2           [123456789] [12345
| [123456789] 5           [123456789] | 1           [123456789] [123456789] | [123456789] [123456789] [12345
| [123456789] [123456789] [123456789] | 8           [123456789] 6           | [123456789] [123456789] [12345
+-------------------------------------+-------------------------------------+------------------------------
```

2. If a cell contains a digit, remove that digit from the list of the possible digits from all its neighboring cells. Neighboring cells are the other cells in the given cell's row, column and sub-grid. For example, the grid after removing the fixed value `4` of the row-2-column-1 cell from its neighboring cells:

```
+-------------------------------------+-------------------------------------+------------------------------
| [123 56789] [123 56789] [123 56789] | [123456789] [123456789] [123456789] | [123456789] 1           [12345
| 4           [123 56789] [123 56789] | [123 56789] [123 56789] [123 56789] | [123 56789] [123 56789] [123 5
| [123 56789] 2           [123 56789] | [123456789] [123456789] [123456789] | [123456789] [123456789] [12345
+-------------------------------------+-------------------------------------+------------------------------
| [123 56789] [123456789] [123456789] | [123456789] 5           [123456789] | 4           [123456789] 7
| [123 56789] [123456789] 8           | [123456789] [123456789] [123456789] | 3           [123456789] [12345
| [123 56789] [123456789] 1           | [123456789] 9           [123456789] | [123456789] [123456789] [12345
+-------------------------------------+-------------------------------------+------------------------------
| 3           [123456789] [123456789] | 4           [123456789] [123456789] | 2           [123456789] [12345
| [123 56789] 5           [123456789] | 1           [123456789] [123456789] | [123456789] [123456789] [12345
| [123 56789] [123456789] [123456789] | 8           [123456789] 6           | [123456789] [123456789] [12345
+-------------------------------------+-------------------------------------+------------------------------
```

3. Repeat the previous step for all the cells that are have been solved (or *fixed*), either pre-filled or filled in the previous iteration of the solution. For example, the grid after removing all fixed values from all non-fixed cells:

```
+-------------------------------------+-------------------------------------+------------------------------
| [    56789] [  3  6789] [  3 567 9] | [ 23 567 9] [ 234 678 ] [ 2345 789] | [    56789] 1           [ 2345
| 4           [1 3  6789] [  3 567 9] | [ 23 567 9] [123  678 ] [123 5 789] | [    56789] [ 23 56789] [ 23 5
| [1   56789] 2           [  3 567 9] | [  3 567 9] [1 34 678 ] [1 345 789] | [    56789] [  3456789] [  345
+-------------------------------------+-------------------------------------+------------------------------
| [ 2   6  9] [  3  6  9] [ 23  6  9] | [ 23  6   ] 5           [123    8 ] | 4           [ 2   6 89] 7
| [ 2  567 9] [   4 67 9] 8           | [ 2   67  ] [12 4 67  ] [12 4  7  ] | 3           [ 2  56  9] [12  5
| [ 2  567  ] [  34 67  ] 1           | [ 23  67  ] 9           [ 234  78 ] | [    56 8 ] [ 2  56 8 ] [ 2  5
+-------------------------------------+-------------------------------------+------------------------------
| 3           [1    6789] [     67 9] | 4           7           [    5 7 9] | 2           [    56789] [1   5
| [ 2   6789] 5           [ 2 4 67 9] | 1           [ 23   7  ] [ 23   7 9] | [     6789] [  34 6789] [  34
| [12    7 9] [1  4  7 9] [ 2 4  7 9] | 8           [ 23   7  ] 6           | [1   5 7 9] [  345 7 9] [1 345
+-------------------------------------+-------------------------------------+------------------------------
```

4. Continue till the grid *settles* , that is, there are no more changes in the possibilities of any cells. For example, the settled grid for the current iteration:

```
+-------------------------------------+-------------------------------------+------------------------------
| [    56789] [  3  6789] [  3 567 9] | [ 23 567 9] [ 234 6 8 ] [ 2345 789] | [    56789] 1           [ 2345
| 4           [1 3  6789] [  3 567 9] | [ 23 567 9] [123  6 8 ] [123 5 789] | [    56789] [ 23 56789] [ 23 5
| [1   56789] 2           [  3 567 9] | [  3 567 9] [1 34 6 8 ] [1 345 789] | [    56789] [  3456789] [  345
+-------------------------------------+-------------------------------------+------------------------------
| [ 2   6  9] [  3  6  9] [ 23  6  9] | [ 23  6   ] 5           [123    8 ] | 4           [ 2   6 89] 7
| [ 2  567 9] [   4 67 9] 8           | [ 2   67  ] [12 4 6   ] [12 4  7  ] | 3           [ 2  56  9] [12  5
| [ 2  567  ] [  34 67  ] 1           | [ 23  67  ] 9           [ 234  78 ] | [    56 8 ] [ 2  56 8 ] [ 2  5
+-------------------------------------+-------------------------------------+------------------------------
| 3           [1    6 89] [     6  9] | 4           7           [    5   9] | 2           [    56 89] [1   5
| [ 2   6789] 5           [ 2 4 67 9] | 1           [ 23      ] [ 23     9] | [     6789] [  34 6789] [  34
| [12    7 9] [1  4  7 9] [ 2 4  7 9] | 8           [ 23      ] 6           | [1   5 7 9] [  345 7 9] [1 345
+-------------------------------------+-------------------------------------+------------------------------
```

5. Once the grid settles, choose one of the non-fixed cells following some strategy. Select one of the digits from all the possibilities of the cell, and fix (assume) the cell to have that digit. Go back to step 1 and repeat.
6. The elimination of possibilities may result in inconsistencies. For example, you may end up with a cell with no possibilities. In such a case, discard that branch of solution, and backtrack to last point where you fixed a cell. Choose a different possibility to fix and repeat.
7. If at any point the grid is completely filled, you've found the solution!
8. If you exhaust all branches of the solution then the puzzle is unsolvable. This can happen if it starts with cells pre-filled wrongly.

This algorithm is actually a Depth-First Search on the state space of the grid configurations. It guarantees to either find a solution or prove a puzzle to be unsolvable.

Setting up

We start with writing types to represent the cells and the grid:

```haskell
data Cell = Fixed Int | Possible [Int] deriving (Show, Eq)
type Row  = [Cell]
type Grid = [Row]
```

A cell is either fixed with a particular digit or has a set of digits as possibilities. So it is natural to represent it as a sum type with `Fixed` and `Possible` constructors. A row is a list of cells and a grid is a list of rows.

We'll take the input puzzle as a string of 81 characters representing the cells, left-to-right and top-to-bottom. An example is:

```
.......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........8.6...
```

Here, `.` represents an non-filled cell. Let's write a function to read this input and parse it to our `Grid` data structure:

```haskell
readGrid :: String -> Maybe Grid
readGrid s
  | length s == 81 = traverse (traverse readCell) . Data.List.Split.chunksOf 9 $ s
  | otherwise      = Nothing
  where
    readCell '.' = Just $ Possible [1..9]
    readCell c
      | Data.Char.isDigit c && c > '0' = Just . Fixed . Data.Char.digitToInt $ c
      | otherwise = Nothing
```

`readGrid` return a `Just grid` if the input is correct, else it returns a `Nothing`. It parses a `.` to a `Possible` cell with all digits as possibilities, and a digit char to a `Fixed` cell with that digit. Let's try it out in the *REPL*:

```
*Main> Just grid = readGrid ".......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........
*Main> mapM_ print grid
[Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,
[Fixed 4,Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible
[Possible [1,2,3,4,5,6,7,8,9],Fixed 2,Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible
[Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,
[Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Fixed 8,Possible [1,2,3,4,5,6,7,8,9],Possible
[Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Fixed 1,Possible [1,2,3,4,5,6,7,8,9],Fixed 9,
[Fixed 3,Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Fixed 4,Possible [1,2,3,4,5,6,7,8,9],
[Possible [1,2,3,4,5,6,7,8,9],Fixed 5,Possible [1,2,3,4,5,6,7,8,9],Fixed 1,Possible [1,2,3,4,5,6,7,8,9],
[Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Fixed 8,Possible
```

The output is a bit unreadable but correct. We can write a few functions to clean it up:

```haskell
showGrid :: Grid -> String
showGrid = unlines . map (unwords . map showCell)
  where
    showCell (Fixed x) = show x
    showCell _ = "."

showGridWithPossibilities :: Grid -> String
showGridWithPossibilities = unlines . map (unwords . map showCell)
  where
    showCell (Fixed x)     = show x ++ "          "
    showCell (Possible xs) =
      (++ "]")
      . Data.List.foldl' (\acc x -> acc ++ if x `elem` xs then show x else " ") "["
      $ [1..9]
```

Back to the *REPL* again:

```
*Main> Just grid = readGrid
".......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........8.6..."
*Main> putStrLn $ showGrid grid
. . . . . . . 1 .
4 . . . . . . . .
. 2 . . . . . . .
. . . . 5 . 4 . 7
. . 8 . . . 3 . .
. . 1 . 9 . . . .
3 . . 4 . . 2 . .
. 5 . 1 . . . . .
. . . 8 . 6 . . .
```

```
*Main> putStrLn $ showGridWithPossibilities grid
[123456789] [123456789] [123456789] [123456789] [123456789] [123456789] [123456789] 1           [1234567
4           [123456789] [123456789] [123456789] [123456789] [123456789] [123456789] [123456789] [1234567
[123456789] 2           [123456789] [123456789] [123456789] [123456789] [123456789] [123456789] [1234567
[123456789] [123456789] [123456789] [123456789] 5           [123456789] 4           [123456789] 7
[123456789] [123456789] 8           [123456789] [123456789] [123456789] 3           [123456789] [1234567
[123456789] [123456789] 1           [123456789] 9           [123456789] [123456789] [123456789] [1234567
3           [123456789] [123456789] 4           [123456789] [123456789] 2           [123456789] [1234567
[123456789] 5           [123456789] 1           [123456789] [123456789] [123456789] [123456789] [1234567
[123456789] [123456789] [123456789] 8           [123456789] 6           [123456789] [123456789] [1234567
```

The output is more readable now. We see that, at the start, all the non-filled cells have all the digits as possible values. We'll use these functions for debugging as we go forward. We can now start solving the puzzle.

Pruning the Cells

We can remove the digits of fixed cells from their neighboring cells, one cell as a time. But, it is faster to find all the fixed digits in a row of cells and remove them from the possibilities of all the non-fixed cells of the row, at once. Then we can repeat this *pruning* step for all the rows of the grid (and columns and sub-grids too! We'll see how).

```haskell
pruneCells :: [Cell] -> Maybe [Cell]
pruneCells cells = traverse pruneCell cells
  where
    fixeds = [x | Fixed x <- cells]

    pruneCell (Possible xs) = case xs Data.List.\\ fixeds of
      []  -> Nothing
      [y] -> Just $ Fixed y
      ys  -> Just $ Possible ys
    pruneCell x = Just x
```

`pruneCells` prunes a list of cells as described before. We start with finding the fixed digits in the list of cells. Then we go over each non-fixed cells, removing the fixed digits we found, from their possible values. Two special cases arise:

- If pruning results in a cell with no possible digits, it is a sign that this branch of search has no solution and hence, we return a `Nothing` in that case.
- If only one possible digit remains after pruning, then we turn that cell into a fixed cell with that digit.

We use the `traverse` function for pruning the cells so that a `Nothing` resulting from pruning one cell propagates to the entire list.

Let's take it for a spin in the *REPL*:

```
*Main> Just grid = readGrid "6......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........8
*Main> putStr $ showGridWithPossibilities $ [head grid] -- first row of the grid
6          [123456789] [123456789] [123456789] [123456789] [123456789] [123456789] 1           [12345678
*Main> putStr $ showGridWithPossibilities [fromJust $ pruneCells $ head grid] -- same row after pruning
6          [ 2345 789] [ 2345 789] [ 2345 789] [ 2345 789] [ 2345 789] [ 2345 789] 1           [ 2345 78
```

It works! `6` and `1` are removed from the possibilities of the other cells. Now we are ready for …

Pruning the Grid

Pruning a grid requires us to prune each row, each column and each sub-grid. Let's try to solve it in the *REPL* first:

```
*Main> Just grid = readGrid "6......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........
*Main> Just grid' = traverse pruneCells grid
*Main> putStr $ showGridWithPossibilities grid'
6           [ 2345 789] [ 2345 789] [ 2345 789] [ 2345 789] [ 2345 789] [ 2345 789] 1           [ 2345 7
4           [123 56789] [123 56789] [123 56789] [123 56789] [123 56789] [123 56789] [123 56789] [123 567
[1 3456789] 2           [1 3456789] [1 3456789] [1 3456789] [1 3456789] [1 3456789] [1 3456789] [1 34567
[123  6 89] [123  6 89] [123  6 89] [123  6 89] 5           [123  6 89] 4           [123  6 89] 7
[12 4567 9] [12 4567 9] 8           [12 4567 9] [12 4567 9] [12 4567 9] 3           [12 4567 9] [12 4567
[ 2345678 ] [ 2345678 ] 1           [ 2345678 ] 9           [ 2345678 ] [ 2345678 ] [ 2345678 ] [ 234567
3           [1   56789] [1   56789] 4           [1   56789] [1   56789] 2           [1   56789] [1   567
[ 234 6789] 5           [ 234 6789] 1           [ 234 6789] [ 234 6789] [ 234 6789] [ 234 6789] [ 234 67
[12345 7 9] [12345 7 9] [12345 7 9] 8           [12345 7 9] 6           [12345 7 9] [12345 7 9] [12345 7
```

By `traverse`-ing the grid with `pruneCells`, we are able to prune each row, one-by-one. Since pruning a row doesn't affect another row, we don't have to pass the resulting rows between each pruning step. That is to say, `traverse` is enough for us, we don't need `foldl` here.

How do we do the same thing for columns now? Since our representation for the grid is rows-first, we first need to convert it to a columns-first representation. Luckily, that's what `Data.List.transpose` function does:

```
*Main> Just grid = readGrid
"693784512487512936125963874932651487568247391741398625319475268856129743274836159"
*Main> putStr $ showGrid grid
6 9 3 7 8 4 5 1 2
4 8 7 5 1 2 9 3 6
1 2 5 9 6 3 8 7 4
9 3 2 6 5 1 4 8 7
5 6 8 2 4 7 3 9 1
7 4 1 3 9 8 6 2 5
3 1 9 4 7 5 2 6 8
8 5 6 1 2 9 7 4 3
2 7 4 8 3 6 1 5 9
*Main> putStr $ showGrid $ Data.List.transpose grid
6 4 1 9 5 7 3 8 2
9 8 2 3 6 4 1 5 7
3 7 5 2 8 1 9 6 4
7 5 9 6 2 3 4 1 8
8 1 6 5 4 9 7 2 3
4 2 3 1 7 8 5 9 6
5 9 8 4 3 6 2 7 1
1 3 7 8 9 2 6 4 5
2 6 4 7 1 5 8 3 9
```

Pruning columns is easy now:

```
*Main> Just grid = readGrid "6......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........
*Main> Just grid' = fmap Data.List.transpose . traverse pruneCells . Data.List.transpose $ grid
*Main> putStr $ showGridWithPossibilities grid'
6           [1 34 6789] [ 234567 9] [ 23 567 9] [1234 678 ] [12345 789] [1   56789] 1           [123456
4           [1 34 6789] [ 234567 9] [ 23 567 9] [1234 678 ] [12345 789] [1   56789] [ 23456789] [123456
[12  5 789] 2           [ 234567 9] [ 23 567 9] [1234 678 ] [12345 789] [1   56789] [ 23456789] [123456
[12  5 789] [1 34 6789] [ 234567 9] [ 23 567 9] 5           [12345 789] 4           [ 23456789] 7
[12  5 789] [1 34 6789] 8           [ 23 567 9] [1234 678 ] [12345 789] 3           [ 23456789] [123456
[12  5 789] [1 34 6789] 1           [ 23 567 9] 9           [12345 789] [1   56789] [ 23456789] [123456
3           [1 34 6789] [ 234567 9] 4           [1234 678 ] [12345 789] 2           [ 23456789] [123456
[12  5 789] 5           [ 234567 9] 1           [1234 678 ] [12345 789] [1   56789] [ 23456789] [123456
[12  5 789] [1 34 6789] [ 234567 9] 8           [1234 678 ] 6           [1   56789] [ 23456789] [123456
```

First, we transpose the grid to convert the columns into rows. Then, we prune the rows by `traverse`-ing `pruneCells` over them. And finally, we turn the rows back into columns by `transpose`-ing the grid back again. The last `transpose` needs to be `fmap`-ped because `traverse pruneCells` returns a `Maybe`.

Pruning sub-grids is a bit trickier. Following the same idea as pruning columns, we need two functions to transform the sub-grids into rows and back. Let's write the first one:

```haskell
subGridsToRows :: Grid -> Grid
subGridsToRows =
  concatMap (\rows -> let [r1, r2, r3] = map (Data.List.Split.chunksOf 3) rows
                      in zipWith3 (\a b c -> a ++ b ++ c) r1 r2 r3)
  . Data.List.Split.chunksOf 3
```

And try it out:

```
*Main> Just grid = readGrid
"693784512487512936125963874932651487568247391741398625319475268856129743274836159"
*Main> putStr $ showGrid grid
6 9 3 7 8 4 5 1 2
4 8 7 5 1 2 9 3 6
1 2 5 9 6 3 8 7 4
9 3 2 6 5 1 4 8 7
5 6 8 2 4 7 3 9 1
7 4 1 3 9 8 6 2 5
3 1 9 4 7 5 2 6 8
8 5 6 1 2 9 7 4 3
2 7 4 8 3 6 1 5 9
*Main> putStr $ showGrid $ subGridsToRows grid
6 9 3 4 8 7 1 2 5
7 8 4 5 1 2 9 6 3
5 1 2 9 3 6 8 7 4
9 3 2 5 6 8 7 4 1
6 5 1 2 4 7 3 9 8
4 8 7 3 9 1 6 2 5
3 1 9 8 5 6 2 7 4
4 7 5 1 2 9 8 3 6
2 6 8 7 4 3 1 5 9
```

You can go over the code and the output and make yourself sure that it works. Also, it turns out that we don’t need to write the back-transform function. `subGridsToRows` is its own back-transform:

```
*Main> putStr $ showGrid grid
6 9 3 7 8 4 5 1 2
4 8 7 5 1 2 9 3 6
1 2 5 9 6 3 8 7 4
9 3 2 6 5 1 4 8 7
5 6 8 2 4 7 3 9 1
7 4 1 3 9 8 6 2 5
3 1 9 4 7 5 2 6 8
8 5 6 1 2 9 7 4 3
2 7 4 8 3 6 1 5 9
*Main> putStr $ showGrid $ subGridsToRows $ subGridsToRows $ grid
6 9 3 7 8 4 5 1 2
4 8 7 5 1 2 9 3 6
1 2 5 9 6 3 8 7 4
9 3 2 6 5 1 4 8 7
5 6 8 2 4 7 3 9 1
7 4 1 3 9 8 6 2 5
3 1 9 4 7 5 2 6 8
8 5 6 1 2 9 7 4 3
2 7 4 8 3 6 1 5 9
```

Nice! Now writing the sub-grid pruning function is easy:

```
*Main> Just grid = readGrid "6......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........
*Main> Just grid' = fmap subGridsToRows . traverse pruneCells . subGridsToRows $ grid
*Main> putStr $ showGridWithPossibilities grid'
6           [1 3 5 789] [1 3 5 789] [123456789] [123456789] [123456789] [ 23456789] 1           [ 234567
4           [1 3 5 789] [1 3 5 789] [123456789] [123456789] [123456789] [ 23456789] [ 23456789] [ 234567
[1 3 5 789] 2           [1 3 5 789] [123456789] [123456789] [123456789] [ 23456789] [ 23456789] [ 234567
[ 234567 9] [ 234567 9] [ 234567 9] [1234 678 ] 5           [1234 678 ] 4           [12  56 89] 7
[ 234567 9] [ 234567 9] 8           [1234 678 ] [1234 678 ] [1234 678 ] 3           [12  56 89] [12  56
[ 234567 9] [ 234567 9] 1           [1234 678 ] 9           [1234 678 ] [12  56 89] [12  56 89] [12  56
3           [12 4 6789] [12 4 6789] 4           [ 23 5 7 9] [ 23 5 7 9] 2           [1 3456789] [1 34567
[12 4 6789] 5           [12 4 6789] 1           [ 23 5 7 9] [ 23 5 7 9] [1 3456789] [1 3456789] [1 34567
[12 4 6789] [12 4 6789] [12 4 6789] 8           [ 23 5 7 9] 6           [1 3456789] [1 3456789] [1 34567
```

It works well. Now we can string together these three steps to prune the entire grid. We also have to make sure that result of pruning each step is fed into the next step. This is so that the fixed cells created into one step cause more pruning in the further steps. We use monadic bind (>>=) for that. Here’s the final code:

```haskell
pruneGrid' :: Grid -> Maybe Grid
pruneGrid' grid =
  traverse pruneCells grid
  >>= fmap Data.List.transpose . traverse pruneCells . Data.List.transpose
  >>= fmap subGridsToRows . traverse pruneCells . subGridsToRows
```

And the test:

```
*Main> Just grid = readGrid "6......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........
*Main> Just grid' = pruneGrid' grid
*Main> putStr $ showGridWithPossibilities grid'
6           [  3   789] [  3 5 7 9] [ 23 5 7 9] [ 234  78 ] [ 2345 789] [    5 789] 1           [ 2345
4           [1 3   789] [  3 5 7 9] [ 23 567 9] [123  678 ] [123 5 789] [    56789] [ 23 56789] [ 23 56
[1   5 789] 2           [  3 5 7 9] [  3 567 9] [1 34 678 ] [1 345 789] [    56789] [  3456789] [  3456
[ 2      9] [  3  6  9] [ 23  6  9] [ 23  6   ] 5           [123    8 ] 4           [ 2   6 89] 7
[ 2  5 7 9] [   4 67 9] 8           [ 2   67  ] [12 4 67  ] [12 4  7  ] 3           [ 2  56  9] [12   56
[ 2  5 7  ] [  34 67  ] 1           [ 23  67  ] 9           [ 234  78 ] [    56 8 ] [ 2  56 8 ] [ 2   56
3           [1    6789] [     67 9] 4           7           [    5 7 9] 2           [    56789] [1   56
[ 2    789] 5           [ 2 4 67 9] 1           [ 23   7  ] [ 23   7 9] [     6789] [  34 6789] [  34 6
[12    7 9] [1  4  7 9] [ 2 4  7 9] 8           [ 23   7  ] 6           [1   5 7 9] [  345 7 9] [1 345
*Main> putStr $ showGrid grid
6 . . . . . . 1 .
4 . . . . . . . .
. 2 . . . . . . .
. . . . 5 . 4 . 7
. . 8 . . . 3 . .
. . 1 . 9 . . . .
3 . . 4 . . 2 . .
. 5 . 1 . . . . .
. . . 8 . 6 . . .
*Main> putStr $ showGrid grid'
6 . . . . . . 1 .
4 . . . . . . . .
. 2 . . . . . . .
. . . . 5 . 4 . 7
. . 8 . . . 3 . .
. . 1 . 9 . . . .
3 . . 4 7 . 2 . .
. 5 . 1 . . . . .
. . . 8 . 6 . . .
```

We can clearly see the massive pruning of possibilities all around the grid. We also see a `7` pop up in the row-7-column-5 cell. This means that we can prune the grid further, until it settles. If you are familiar with Haskell, you may recognize this as trying to find a fixed point for the `pruneGrid'` function, except in a monadic context. It is simple to implement:

```haskell
pruneGrid :: Grid -> Maybe Grid
pruneGrid = fixM pruneGrid'
  where
    fixM f x = f x >>= \x' -> if x' == x then return x else fixM f x'
```

The crux of this code is the `fixM` function. It takes a monadic function `f` and an initial value, and recursively calls itself till the return value settles. Let's do another round in the *REPL*:

```
*Main> Just grid = readGrid "6......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........
*Main> Just grid' = pruneGrid grid
*Main> putStr $ showGridWithPossibilities grid'
6           [  3   789] [  3 5 7 9] [ 23 5 7 9] [ 234   8 ] [ 2345 789] [    5 789] 1           [ 2345
4           [1 3   789] [  3 5 7 9] [ 23 567 9] [123  6 8 ] [123 5 789] [    56789] [ 23 56789] [ 23 56
[1   5 789] 2           [  3 5 7 9] [  3 567 9] [1 34 6 8 ] [1 345 789] [    56789] [  3456789] [  3456
[ 2      9] [  3  6  9] [ 23  6  9] [ 23  6   ] 5           [123    8 ] 4           [ 2   6 89] 7
[ 2  5 7 9] [   4 67 9] 8           [ 2   67  ] [12 4 6   ] [12 4  7  ] 3           [ 2  56  9] [12  56
[ 2  5 7  ] [  34 67  ] 1           [ 23  67  ] 9           [ 234  78 ] [    56 8 ] [ 2  56 8 ] [ 2  56
3           [1    6 89] [     6  9] 4           7           [    5   9] 2           [    56 89] [1   56
[ 2    789] 5           [ 2 4 67 9] 1           [ 23      ] [ 23     9] [     6789] [  34 6789] [  34 6
[12    7 9] [1  4  7 9] [ 2 4  7 9] 8           [ 23      ] 6           [1   5 7 9] [  345 7 9] [1 345
```

We see that `7` in the row-7-column-5 cell is eliminated from all its neighboring cells. We can't prune the grid anymore. Now it is time to make the choice.

Making the Choice

One the grid is settled, we need to choose a non-fixed cell and make it fixed by assuming one of its possible values. This gives us two grids, next in the state-space of the solution search:

- one which has this chosen cell fixed to this chosen digit, and,
- the other in which the chosen cell has all the other possibilities except the one we chose to fix.

We call this function, `nextGrids`:

```haskell
nextGrids :: Grid -> (Grid, Grid)
nextGrids grid =
  let (i, first@(Fixed _), rest) =
        fixCell
        . Data.List.minimumBy (compare `Data.Function.on` (possibilityCount . snd))
        . filter (isPossible . snd)
        . zip [0..]
        . concat
        $ grid
  in (replace2D i first grid, replace2D i rest grid)
  where
    isPossible (Possible _) = True
    isPossible _            = False

    possibilityCount (Possible xs) = length xs
    possibilityCount (Fixed _)     = 1

    fixCell (i, Possible [x, y]) = (i, Fixed x, Fixed y)
    fixCell (i, Possible (x:xs)) = (i, Fixed x, Possible xs)
    fixCell _                    = error "Impossible case"

    replace2D :: Int -> a -> [[a]] -> [[a]]
    replace2D i v =
      let (x, y) = (i `quot` 9, i `mod` 9) in replace x (replace y (const v))
    replace p f xs = [if i == p then f x else x | (x, i) <- zip xs [0..]]
```

We choose the non-fixed cell with least count of possibilities as the pivot. This strategy make sense intuitively, as with a cell with fewest possibilities, we have the most chance of being right when assuming one. Fixing a non-fixed cell leads to one of the two cases:

a. the cell has only two possible values, resulting in two fixed cells, or,
b. the cell has more than two possible values, resulting in one fixed and one non-fixed cell.

Then all we are left with is replacing the non-fixed cell with its fixed and fixed/non-fixed choices, which we do with some math and some list traversal. A quick check on the *REPL* :

```
*Main> Just grid = readGrid "6......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........
*Main> Just grid' = pruneGrid grid
*Main> putStr $ showGridWithPossibilities grid'
6           [  3   789] [  3 5 7 9] [ 23 5 7 9] [ 234   8 ] [ 2345 789] [    5 789] 1           [ 2345
4           [1 3   789] [  3 5 7 9] [ 23 567 9] [123  6 8 ] [123 5 789] [    56789] [ 23 56789] [ 23 56
[1   5 789] 2           [  3 5 7 9] [  3 567 9] [1 34 6 8 ] [1 345 789] [    56789] [  3456789] [  3456
[ 2      9] [  3  6  9] [ 23  6  9] [ 23  6   ] 5           [123    8 ] 4           [ 2   6 89] 7
[ 2  5 7 9] [   4 67 9] 8           [ 2   67  ] [12 4 6   ] [12 4  7  ] 3           [ 2  56  9] [12  56
[ 2  5 7  ] [  34 67  ] 1           [ 23  67  ] 9           [ 234  78 ] [    56 8 ] [ 2  56 8 ] [ 2  56
3           [1    6 89] [     6  9] 4           7           [    5   9] 2           [    56 89] [1   56
[ 2    789] 5           [ 2 4 67 9] 1           [ 23      ] [ 23     9] [     6789] [  34 6789] [  34 6
[12    7 9] [1  4  7 9] [ 2 4  7 9] 8           [ 23      ] 6           [1   5 7 9] [  345 7 9] [1 345
*Main> -- the row-4-column-1 cell is the first cell with only two possibilities, [2, 9].
*Main> -- it is chosen as the pivot cell to find the next grids.
*Main> (grid1, grid2) = nextGrids grid'
*Main> putStr $ showGridWithPossibilities grid1
6           [  3   789] [  3 5 7 9] [ 23 5 7 9] [ 234   8 ] [ 2345 789] [    5 789] 1           [ 2345
4           [1 3   789] [  3 5 7 9] [ 23 567 9] [123  6 8 ] [123 5 789] [    56789] [ 23 56789] [ 23 56
[1   5 789] 2           [  3 5 7 9] [  3 567 9] [1 34 6 8 ] [1 345 789] [    56789] [  3456789] [  3456
2           [  3  6  9] [ 23  6  9] [ 23  6   ] 5           [123    8 ] 4           [ 2   6 89] 7
[ 2  5 7 9] [   4 67 9] 8           [ 2   67  ] [12 4 6   ] [12 4  7  ] 3           [ 2  56  9] [12  56
[ 2  5 7  ] [  34 67  ] 1           [ 23  67  ] 9           [ 234  78 ] [    56 8 ] [ 2  56 8 ] [ 2  56
3           [1    6 89] [     6  9] 4           7           [    5   9] 2           [    56 89] [1   56
[ 2    789] 5           [ 2 4 67 9] 1           [ 23      ] [ 23     9] [     6789] [  34 6789] [  34 6
[12    7 9] [1  4  7 9] [ 2 4  7 9] 8           [ 23      ] 6           [1   5 7 9] [  345 7 9] [1 345
*Main> putStr $ showGridWithPossibilities grid2
6           [  3   789] [  3 5 7 9] [ 23 5 7 9] [ 234   8 ] [ 2345 789] [    5 789] 1           [ 2345
4           [1 3   789] [  3 5 7 9] [ 23 567 9] [123  6 8 ] [123 5 789] [    56789] [ 23 56789] [ 23 56
[1   5 789] 2           [  3 5 7 9] [  3 567 9] [1 34 6 8 ] [1 345 789] [    56789] [  3456789] [  3456
9           [  3  6  9] [ 23  6  9] [ 23  6   ] 5           [123    8 ] 4           [ 2   6 89] 7
[ 2  5 7 9] [   4 67 9] 8           [ 2   67  ] [12 4 6   ] [12 4  7  ] 3           [ 2  56  9] [12  56
[ 2  5 7  ] [  34 67  ] 1           [ 23  67  ] 9           [ 234  78 ] [    56 8 ] [ 2  56 8 ] [ 2  56
3           [1    6 89] [     6  9] 4           7           [    5   9] 2           [    56 89] [1   56
[ 2    789] 5           [ 2 4 67 9] 1           [ 23      ] [ 23     9] [     6789] [  34 6789] [  34 6
[12    7 9] [1  4  7 9] [ 2 4  7 9] 8           [ 23      ] 6           [1   5 7 9] [  345 7 9] [1 345
```

Solving the Puzzle

We have implemented parts of our algorithm till now. Now we'll put everything together to solve the puzzle. First, we need to know if we are done or have messed up:

```haskell
isGridFilled :: Grid -> Bool
isGridFilled grid = null [ () | Possible _ <- concat grid ]

isGridInvalid :: Grid -> Bool
isGridInvalid grid =
  any isInvalidRow grid
  || any isInvalidRow (Data.List.transpose grid)
  || any isInvalidRow (subGridsToRows grid)
  where
    isInvalidRow row =
      let fixeds         = [x | Fixed x <- row]
          emptyPossibles = [x | Possible x <- row, null x]
      in hasDups fixeds || not (null emptyPossibles)

    hasDups l = hasDups' l []

    hasDups' [] _ = False
    hasDups' (y:ys) xs
      | y `elem` xs = True
      | otherwise   = hasDups' ys (y:xs)
```

`isGridFilled` returns whether a grid is filled completely by checking it for any `Possible` cells. `isGridInvalid` checks if a grid is invalid because it either has duplicate fixed cells in any block or has any non-fixed cell with no possibilities.

Writing the `solve` function is almost trivial now:

```haskell
solve :: Grid -> Maybe Grid
solve grid = pruneGrid grid >>= solve'
  where
    solve' g
      | isGridInvalid g = Nothing
      | isGridFilled g  = Just g
      | otherwise       =
          let (grid1, grid2) = nextGrids g
          in solve grid1 <|> solve grid2
```

We prune the grid as before and pipe it to the helper function `solve'`. `solve'` bails with a `Nothing` if the grid is invalid, or returns the solved grid if it is filled completely. Otherwise, it finds the next two grids in the search tree and solves them recursively with backtracking by calling the `solve` function. Backtracking here is implemented by the using the `Alternative` (`<|>`) implementation of the `Maybe` type[2]. It takes the second branch in the computation if the first branch returns a `Nothing`.

Whew! That took us long. Let's put it to the final test now:

```
*Main> Just grid =
  readGrid "6......1.4.........2...........5.4.7..8...3....1.9....3..4..2...5.1........8.6..."
*Main> putStr $ showGrid grid
6 . . . . . . 1 .
4 . . . . . . . .
. 2 . . . . . . .
. . . . 5 . 4 . 7
. . 8 . . . 3 . .
. . 1 . 9 . . . .
3 . . 4 . . 2 . .
. 5 . 1 . . . . .
. . . 8 . 6 . . .
*Main> Just grid' = solve grid
*Main> putStr $ showGrid grid'
6 9 3 7 8 4 5 1 2
4 8 7 5 1 2 9 3 6
1 2 5 9 6 3 8 7 4
9 3 2 6 5 1 4 8 7
5 6 8 2 4 7 3 9 1
7 4 1 3 9 8 6 2 5
3 1 9 4 7 5 2 6 8
8 5 6 1 2 9 7 4 3
2 7 4 8 3 6 1 5 9
```

It works! Let’s put a quick `main` wrapper around `solve` to call it from the command line:

```haskell
main :: IO ()
main = do
  inputs <- lines <$> getContents
  Control.Monad.forM_ inputs $ \input ->
    case readGrid input of
      Nothing   -> putStrLn "Invalid input"
      Just grid -> case solve grid of
        Nothing    -> putStrLn "No solution found"
        Just grid' -> putStrLn $ showGrid grid'
```

And now, we can invoke it from the command line:

```
$ echo ".......12.5.4............3.7..6..4....1..........8....92....8.....51.7.......3..." |
stack exec sudoku
3 6 4 9 7 8 5 1 2
1 5 2 4 3 6 9 7 8
8 7 9 1 2 5 6 3 4
7 3 8 6 5 1 4 2 9
6 9 1 2 4 7 3 8 5
2 4 5 3 8 9 1 6 7
9 2 3 7 6 4 8 5 1
4 8 6 5 1 2 7 9 3
5 1 7 8 9 3 2 4 6
```

And, we are done.

If you want to play with different puzzles, the file here lists some of the toughest ones. Let's run[3] some of them through our program to see how fast it is:

```
$ head -n100 sudoku17.txt | time stack exec sudoku
... output omitted ...
      116.70 real       198.09 user        94.46 sys
```

It took about 117 seconds to solve a hundred puzzles, so, about 1.2 seconds per puzzle. This is pretty slow but we'll get around to making it faster in the subsequent posts.

Conclusion

In this rather verbose article, we learned how to write a simple Sudoku solver in Haskell step-by-step. In the later parts of this series, we'll delve into profiling the solution and figuring out better algorithms and data structures to solve Sudoku more efficiently. The code till now is available here. Discuss this post on r/haskell or leave a comment.

Footnotes

1. This exercise was originally done as a part of the Haskell classes I taught at nilenso.↩
2. `Alternative` implementation of `Maybe`:

   ```haskell
   instance Alternative Maybe where
     empty = Nothing
     Nothing <|> r = r
     l       <|> _ = l
   ```

   ↩
3. All the runs were done on my MacBook Pro from 2014 with 2.2 GHz Intel Core i7 CPU and 16 GB memory.↩