Fast Sudoku Solver in Haskell #2: A 200x Faster Solution

✏ July 11, 2018

🕐 A sixteen minute read

❥ Tags: haskell, sudoku, programming, puzzle, nilenso

I n the first part of this series of posts, we wrote a simple Sudoku solver in Haskell. It used a constraint satisfaction algorithm with backtracking. The solution worked well but was very slow. In this post, we are going to improve it and make it **fast**.

This is the second 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

Quick Recap

Sudoku is a number placement puzzle. It consists of a 9x9 grid which is to be filled with digits from 1 to 9 such that each row, each column and each of the nine 3x3 sub-grids contain all the digits. Some of the cells of the grid come pre-filled and the player has to fill the rest.

In the previous post, we implemented a simple Sudoku solver without paying much attention to its performance characteristics. We ran[1] some of 17-clue puzzles[2] through our program to see how fast it was:

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

So, it took about 117 seconds to solve one hundred puzzles. At this speed, it would take about 16 hours to solve all the 49151 puzzles contained in the file. This is way too slow. We need to find ways to make it faster. Let's go back to the drawing board.

Constraints and Corollaries

In a Sudoku puzzle, we have a partially filled 9x9 grid which we have to fill completely while following the constraints of the game.

```
+-------+-------+-------+
| . . . | . . . | . 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
```

Earlier, we followed a simple pruning algorithm which removed all the solved (or *fixed*) digits from neighbours of the fixed cells. We repeated the pruning till the fixed and non-fixed values in the grid stopped changing (or the grid *settled*). Here's an example of a grid before pruning:

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

And here's the same grid when it settles after repeated pruning:

```
+-----------------------------------+-----------------------------------+----------------------------
-------+
| [    56789] [ 3  6789] [ 3 567 9] | [ 23 567 9] [ 234 6 8 ] [ 2345 789] | [    56789] 1          [
23456 89] |
| 4          [1 3  6789] [ 3 567 9] | [ 23 567 9] [123  6 8 ] [123 5 789] | [    56789] [ 23 56789] [ 23
56 89] |
| [1    56789] 2          [ 3 567 9] | [ 3 567 9] [1 34 6 8 ] [1 345 789] | [    56789] [ 3456789] [
3456 89] |
+-----------------------------------+-----------------------------------+----------------------------
-------+
| [ 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
56  9] |
| [ 2  567 ] [  34 67 ] 1          | [ 23  67 ] 9          [ 234  78 ] | [    56 8 ] [ 2  56 8 ] [ 2
56 8 ] |
+-----------------------------------+-----------------------------------+----------------------------
-------+
| 3          [1    6 89] [    6  9] | 4          7          [    5  9] | 2          [    56 89] [1
56 89] |
| [ 2   6789] 5          [ 2 4 67 9] | 1          [ 23    ] [ 23    9] | [    6789] [ 34 6789] [  34
6 89] |
| [12    7 9] [1 4 7 9] [ 2 4  7 9] | 8          [ 23    ] 6          | [1   5 7 9] [ 345 7 9] [1
345   9] |
+-----------------------------------+-----------------------------------+----------------------------
-------+
```

We see how the possibilities conflicting with the fixed values are removed. We also see how some of the non-fixed cells turn into fixed ones as all their other possible values get eliminated.

This simple strategy follows directly from the constraints of Sudoku. But, are there more complex strategies which are implied indirectly?

Singles, Twins and Triplets

Let's have a look at this sample row captured from a solution in progress:

```
+-----------------------------------+-----------------------------------+---------------------------
-------+
| 4          [ 2   6 89] 7          | 3          [ 2 56  9] [12 56  9] | [   56 8 ] [ 2  56 8 ] [ 2
56 8 ] |
+-----------------------------------+-----------------------------------+---------------------------
-------+
```

Notice how the sixth cell is the only one with 1 as a possibility in it. It is obvious that we should fix the sixth cell to 1 as we cannot place 1 in any other cell in the row. Let's call this the *Singles* [3] scenario.

But, our current solution will not fix the sixth cell to 1 till one of these cases arise:

   a. all other possibilities of the cell are pruned away, or,
   b. the cell is chosen as pivot in the nextGrids function and 1 is chosen as the value to fix.

This may take very long and lead to a longer solution time. Let's assume that we recognize the Singles scenario while pruning cells and fix the cell to 1 right then. That would cut down the search tree by a lot and make the solution much faster.

It turns out, we can generalize this pattern. Let's check out this sample row from middle of a solution:

```
+-----------------------------------+-----------------------------------+---------------------------
-------+
| [1  4    9] 3          [1  4567 9] | [1  4   89] [1  4 6 89] [1  4 6 89] | [1  4   89] 2          [1
456789] |
+-----------------------------------+-----------------------------------+---------------------------
-------+
```

It is a bit difficult to notice with the naked eye but there's something special here too. The digits 5 and 7 occur only in the third and the ninth cells. Though they are accompanied by other digits in those cells, they are not present in any other cells. This means, we can place 5 and 7 either in the third or the ninth cell and no other cells. This implies that we can prune the third and ninth cells to have only 5 and 7 like this:

```
+-----------------------------------+-----------------------------------+---------------------------
-------+
| [1  4    9] 3          [   5 7  ] | [1  4   89] [1  4 6 89] [1  4 6 89] | [1  4   89] 2          [
5 7  ] |
+-----------------------------------+-----------------------------------+---------------------------
-------+
```

This is the *Twins* scenario. As we can imagine, this pattern extends to groups of three digits and beyond. When three digits can be found only in three cells in a block, it is the *Triplets* scenario, as in the example below:

```
+-----------------------------------+-----------------------------------+---------------------------
-------+
| [   45 7  ] [   45 7  ] [   5 7  ] | 2          [ 3 5  89] 6          | 1          [  34   89] [  34
89] |
+-----------------------------------+-----------------------------------+---------------------------
-------+
```

In this case, the triplet digits are 3, 8 and 9. And as before, we can prune the block by fixing these digits in their cells:

```
+-----------------------------------+-----------------------------------+---------------------------
-------+
| [   45 7  ] [   45 7  ] [   5 7  ] | 2          [ 3    89] 6          | 1          [  3    89] [  3
89] |
+-----------------------------------+-----------------------------------+---------------------------
-------+
```

Let's call these three scenarios *Exclusives* in general.

We can extend this to *Quadruplets* scenario and further. But such scenarios occur rarely in a 9x9 Sudoku puzzle. Trying to find them may end up being more computationally expensive than the benefit we may get in solution time speedup by finding them.

Now that we have discovered these new strategies to prune cells, let's implement them in Haskell.

A Little Forward, a Little Backward

We can implement the three new strategies to prune cells as one function for each. However, we can actually implement all these strategies in a single function. But, this function is a bit more complex than the previous pruning function. So first, let's try to understand its working using tables. Let's take this sample row:

```
+-----------------------------------+-----------------------------------+----------------------------
-------+
| [   4 6   9] 1           5         | [      6   9] 7           [ 23  6 89] | [      6   9] [ 23  6 89] [ 23
 6 89] |
+-----------------------------------+-----------------------------------+----------------------------
-------+
```

First, we make a table mapping the digits to the cells in which they occur, excluding the fixed cells:

Digit	Cells
2	6, 8, 9
3	6, 8, 9
4	1
6	1, 4, 6, 7, 8, 9
8	6, 8, 9
9	1, 4, 6, 7, 8, 9

Then, we flip this table and collect all the digits that occur in the same set of cells:

Cells	Digits
1	4
6, 8, 9	2, 3, 8
1, 4, 6, 7, 8, 9	6, 9

And finally, we remove the rows of the table in which the count of the cells is not the same as the count of the digits:

Cells	Digits
1	4
6, 8, 9	2, 3, 8

Voilà! We have found a Single 4 and a set of Triplets 2, 3 and 8. You can go over the puzzle row and verify that this indeed is the case.

Translating this logic to Haskell is quite easy now:

```haskell
1   isPossible :: Cell -> Bool
2   isPossible (Possible _) = True
3   isPossible _            = False
4
5   exclusivePossibilities :: [Cell] -> [[Int]]
6   exclusivePossibilities row =
7     -- input
8     row
9     -- [Possible [4,6,9], Fixed 1, Fixed 5, Possible [6,9], Fixed 7, Possible [2,3,6,8,9],
10    -- Possible [6,9], Possible [2,3,6,8,9], Possible [2,3,6,8,9]]
11
12    -- step 1
13    & zip [1..9]
14    -- [(1,Possible [4,6,9]),(2,Fixed 1),(3,Fixed 5),(4,Possible [6,9]),(5,Fixed 7),
15    -- (6,Possible [2,3,6,8,9]),(7,Possible [6,9]),(8,Possible [2,3,6,8,9]),
16    -- (9,Possible [2,3,6,8,9])]
17
18    -- step 2
19    & filter (isPossible . snd)
20    -- [(1,Possible [4,6,9]),(4,Possible [6,9]),(6,Possible [2,3,6,8,9]),
21    -- (7,Possible [6,9]), (8,Possible [2,3,6,8,9]),(9,Possible [2,3,6,8,9])]
22
23    -- step 3
24    & Data.List.foldl'
25        (\acc ~(i, Possible xs) ->
26          Data.List.foldl' (\acc' x -> Map.insertWith prepend x [i] acc') acc xs)
27        Map.empty
28    -- fromList [(2,[9,8,6]),(3,[9,8,6]),(4,[1]),(6,[9,8,7,6,4,1]),(8,[9,8,6]),
29    -- (9,[9,8,7,6,4,1])]
30
31    -- step 4
32    & Map.filter ((< 4) . length)
33    -- fromList [(2,[9,8,6]),(3,[9,8,6]),(4,[1]),(8,[9,8,6])]
34
35    -- step 5
36    & Map.foldlWithKey'(\acc x is -> Map.insertWith prepend is [x] acc) Map.empty
37    -- fromList [([1],[4]),([9,8,6],[8,3,2])]
38
39    -- step 6
40    & Map.filterWithKey (\is xs -> length is == length xs)
41    -- fromList [([1],[4]),([9,8,6],[8,3,2])]
42
43    -- step 7
44    & Map.elems
45    -- [[4],[8,3,2]]
46    where
47      prepend ~[y] ys = y:ys
```

We extract the isPossible function to the top level from the nextGrids function for reuse. Then we write the exclusivePossibilities function which finds the Exclusives in the input row. This function is written using the reverse application operator (&)[4] instead of the usual ($) operator so that we can read it from top to bottom. We also show the intermediate values for a sample input after every step in the function chain.

The nub of the function lies in step 3 (pun intended). We do a nested fold over all the non-fixed cells and all the possible digits in them to compute the map[5] which represents the first table. Thereafter, we filter the map to keep only the entries with length less than four (step 4). Then we flip it to create a new map which represents the second table (step 5). Finally, we filter the flipped map for the entries where the cell count is same as the digit count (step 6) to arrive at the final table. The step 7 just gets the values in the map which is the list of all the Exclusives in the input row.

Pruning the Cells, Exclusively

To start with, we extract some reusable code from the previous pruneCells function and rename it to pruneCellsByFixed:

```
1   makeCell :: [Int] -> Maybe Cell
2   makeCell ys = case ys of
3     []  -> Nothing
4     [y] -> Just $ Fixed y
5     _   -> Just $ Possible ys
6
7   pruneCellsByFixed :: [Cell] -> Maybe [Cell]
8   pruneCellsByFixed cells = traverse pruneCell cells
9     where
10      fixeds = [x | Fixed x <- cells]
11
12      pruneCell (Possible xs) = makeCell (xs Data.List.\\ fixeds)
13      pruneCell x             = Just x
```

Now we write the pruneCellsByExclusives function which uses the exclusivePossibilities function to prune the cells:

```
1   pruneCellsByExclusives :: [Cell] -> Maybe [Cell]
2   pruneCellsByExclusives cells = case exclusives of
3     [] -> Just cells
4     _  -> traverse pruneCell cells
5     where
6       exclusives    = exclusivePossibilities cells
7       allExclusives = concat exclusives
8
9       pruneCell cell@(Fixed _) = Just cell
10      pruneCell cell@(Possible xs)
11        | intersection `elem` exclusives = makeCell intersection
12        | otherwise                      = Just cell
13        where
14          intersection = xs `Data.List.intersect` allExclusives
```

`pruneCellsByExclusives` works exactly as shown in the examples above. We first find the list of Exclusives in the given cells. If there are no Exclusives, there's nothing to do and we just return the cells. If we find any Exclusives, we `traverse` the cells, pruning each cell to only the intersection of the possible digits in the cell and Exclusive digits. That's it! We reuse the `makeCell` function to create a new cell with the intersection.

As the final step, we rewrite the `pruneCells` function by combining both the functions.

```
1  fixM :: (Eq t, Monad m) => (t -> m t) -> t -> m t
2  fixM f x = f x >>= \x' -> if x' == x then return x else fixM f x'
3
4  pruneCells :: [Cell] -> Maybe [Cell]
5  pruneCells cells = fixM pruneCellsByFixed cells >>= fixM pruneCellsByExclusives
```

We have extracted `fixM` as a top level function from the `pruneGrid` function. Just like the `pruneGrid'` function, we need to use monadic bind (>>=) to chain the two pruning steps. We also use `fixM` to apply each step repeatedly till the pruned cells settle[6].

No further code changes are required. It is time to check out the improvements.

Faster than a Speeding Bullet!

Let's build the program and run the exact same number of puzzles as before:

```
$ head -n100 sudoku17.txt | time stack exec sudoku
... output omitted ...
     0.53 real         0.58 user         0.23 sys
```

Woah! It is way faster than before. Let's solve all the puzzles now:

```
$ cat sudoku17.txt | time stack exec sudoku > /dev/null
     282.98 real        407.25 user        109.27 sys
```

So it is took about 283 seconds to solve all the 49151 puzzles. The speedup is about 200x[7]. That's about 5.8 milliseconds per puzzle.

Let's do a quick profiling to see where the time is going:

```
$ stack build --profile
$ head -n1000 sudoku17.txt | stack exec -- sudoku +RTS -p > /dev/null
```

This generates a file named `sudoku.prof` with the profiling results. Here are the top five most time-taking functions (cleaned for brevity):

Cost Center	Source	%time	%alloc
exclusivePossibilities	(49,1)-(62,26)	17.6	11.4
pruneCellsByFixed.pruneCell	(75,5)-(76,36)	16.9	30.8
exclusivePossibilities.\.\	55:38-70	12.2	20.3
fixM.\	13:27-65	10.0	0.0
==	15:56-57	7.2	0.0

Looking at the report, my guess is that a lot of time is going into list operations. Lists are known to be inefficient in Haskell so maybe we should switch to some other data structures?

Update

As per the comment below by Chris Casinghino, I ran both the versions of code without the `-threaded`, `-rtsopts` and `-with-rtsopts=-N` options. The time for previous post's code:

```
$ head -n100 sudoku17.txt | time stack exec sudoku
... output omitted ...
      96.54 real        95.90 user         0.66 sys
```

And the time for this post's code:

```
$ cat sudoku17.txt | time stack exec sudoku > /dev/null
     258.97 real       257.34 user         1.52 sys
```

So, both the versions run about 10% faster without the threading options. I suspect this has something to do with GHC's parallel GC as described in this post. So for now, I'll keep threading disabled.

Conclusion

In this post, we improved upon our simple Sudoku solution from the last time. We discovered and implemented a new strategy to prune cells, and we achieved a 200x speedup. But profiling shows that we still have many possibilities for improvements. We'll work on that and more in the upcoming posts in this series. The code till now is available here. Discuss this post on r/haskell or leave a comment.

Footnotes

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

2. At least 17 cells must be pre-filled in a Sudoku puzzle for it to have a unique solution. So 17-clue puzzles are the most difficult of all puzzles. This paper by McGuire, Tugemann and Civario gives the proof of the same.↩

3. "Single" as in "Single child"↩

4. Reverse application operation is not used much in Haskell. But it is the preferred way of function chaining in some other functional programming languages like Clojure, FSharp, and Elixir.↩

5. We use Data.Map.Strict as the map implementation.↩

6. We need to run `pruneCellsByFixed` and `pruneCellsByExclusives` repeatedly using `fixM` because an unsettled row can lead to wrong solutions.

   Imagine a row which just got a `9` fixed because of `pruneCellsByFixed`. If we don't run the function again, the row may be left with one non-fixed cell with a `9`. When we run this row through `pruneCellsByExclusives`, it'll consider the `9` in the non-fixed cell as a Single and fix it. This will lead to two `9`s in the same row, causing the solution to fail.↵

7. Speedup calculation: 116.7 / 100 * 49151 / 282.98 = 202.7↵