

# Functional Programming Languages Research Paper Summary —— A Haskell Solution to Sudoku Puzzles

Delong Meng

delongmeng@hotmail.com

Paper: Bird, Richard. Functional pearl: A program to solve sudoku.

Journal of functional programming 16.6 (2006): 671-679.

Paper link: <https://www.cs.tufts.edu/~nr/cs257/archive/richard-bird/sudoku.pdf>

Presentation link: [https://mediaspace.illinois.edu/media/t/1\\_axwq3ysn](https://mediaspace.illinois.edu/media/t/1_axwq3ysn)

Code link: [https://github.com/delongmeng/Sudoku\\_Haskell](https://github.com/delongmeng/Sudoku_Haskell)

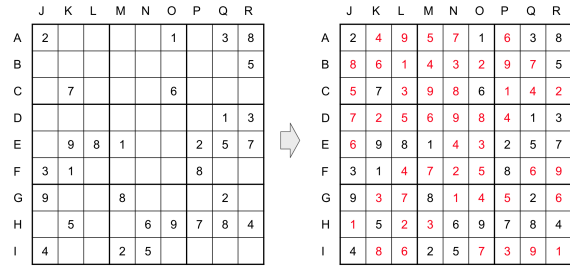
## 1 Introduction

Functional pearls are elegant and instructive examples of functional programming. This paper (Bird, 2006) is a functional pearl that highlights the idea of “wholemeal programming”, which focuses on the entire data structures rather than details of its components, such as coordinate systems and arithmetic operations on subscripts in the sudoku problem. This is a great example to learn and practice equational reasoning and lawful program construction. Here I summarize this paper and what I have learned from it.

## 2 The Problem

A typical sudoku problem provides a 9-by-9 board (81 cells) with some of these cells filled with numbers, and the goal is to fill in the remaining cells so that it can satisfy the rules (Figure 1). The rules are: each of the 9 rows, each of the 9 columns, and each of the 9 3-by-3 boxes (shown in thicker boarders) should contain numbers 1 to 9 (exactly once for each number).

The example problem in Figure 1 is from the manuscript (Bird, 2006), and one can easily solve the problem by hand. For example, in row H, there are only 3 empty cells (HJ, HL, and HM), and one can conclude that these 3 cells should contain these numbers (without knowing the order yet): 1, 2, 3. For the cell HJ, if we look at column J, we can infer that HJ cannot be 2 or 3 because they already exist in that column; thus, HJ has to be 1 (and we can double check that filling in 1 in that cell does not violate the rule for the 3-by-3 box that HJ is in). Now, let us shift our focus to the cell HM, which has to be either 2 or 3, and because 2 is already there in column M, the cell HM has to be 3 (and again it does not cause problem for the box it is in). Then, the cell HL has to be 2. By doing this, we can fill in more and more values into the board step-



	J	K	L	M	N	O	P	Q	R
A	2					1		3	8
B									5
C		7				6			
D								1	3
E		9	8	1				2	5
F	3	1						8	
G	9			8					2
H		5			6	9	7	8	4
I	4			2	5				

	J	K	L	M	N	O	P	Q	R
A	2	4	9	5	7	1	6	3	8
B	8	6	1	4	3	2	9	7	5
C	5	7	3	9	8	6	1	4	2
D	7	2	5	6	9	8	4	1	3
E	6	9	8	1	4	3	2	5	7
F	3	1	4	7	2	5	8	6	9
G	9	3	7	8	1	4	5	2	6
H	1	5	2	3	6	9	7	8	4
I	4	8	6	2	5	7	3	9	1

Figure 1: A typical sudoku problem (left) and its solution (right).

by-step, and eventually solve the whole problem (right side of Figure 1).

## 3 The Program

The goal of this paper is to develop a Haskell program to solve the Sudoku puzzles. This function should be general enough that it can handle any  $N^2 \times N^2$  boards for positive  $N$  (then each box will have the size of  $N \times N$  and the total number of different characters will be  $N$ , instead of the fixed value 9 in our earlier example):

$$\text{sudoku} :: \text{Board} \rightarrow [\text{Board}]$$

### 3.1 Initial solution

The initial version is a version that could at least work and serve as a start point, with no guarantee of efficiency. For any given board that's partially filled as the input (so we also know the  $N$ ), there are 2 simple strategies to get the solutions. One is that we first get a list of all possible completed  $N^2 \times N^2$  sudoku boards, and then filter them using the filled characters in the input board; the second is that we enumerate through all of the possible way of filling the given board and test whether they are valid solutions. We can image that these methods will be much less efficient than the way we as human beings solve sudoku problems (as described

in section 2), but much easier for a computer to perform the steps. This paper went with the 2nd direction and derived the final form of the initial solution as:

$$sudoku = filter\ correct . mcp . choices$$

For details, please refer to the original paper (Bird, 2006), but I'll summarize the main components here. Specifically, this part set up the basic building blocks of this problem as follows:

- *Matrix* is a simple data type of list of lists, and a sudoku *Board* is a *Matrix* of *Char* (for a 9x9 board, any character of "123456789");
- *choices* is a function that takes a partially filled *Board* as an input, and replaces the character (including the blank cells) in each cell with a list of possible fillings (*Choices*);
- *mcp* is a function that generates the matrix cartesian product – basically, it returns a list of all possible completed *Boards* (all ways of filling the blank cells);
- *correct* is a function that check each *Board* whether it is a valid sudoku *Board*, and then *filter* will only keep all of the valid *Boards*;
- Note that *correct* uses 3 very basic helper functions: *rows*, *cols*, and *boxs*. As the names suggest, they extract the rows, columns, and  $N \times N$  boxes of a *Board*;
- An important property of these 3 functions is:  $rows . rows = id$ ,  $cols . cols = id$ ,  $boxs . boxs = id$ .

As I mentioned earlier, this solution is definitely not efficient. For the example board in Figure 1, it will just ignore the fact that the row A already has 1, 2, 3 and 8, and it still fills each of the blank cells in that row with 1-9. How stupid! In fact, for a 9x9 board with 51 empty cells as in our example board, this solution will try all  $9^{51}$  possible boards before it can finish!

### 3.2 Pruning the choices

Next, the paper introduced a key part to improve the solution, and that is the *prune* function. Now the *sudoku* function becomes:

$$sudoku = filter\ correct . mcp . prune . choices$$

where the definition of *prune* is:

$$\begin{aligned} prune = \\ &pruneBy\ boxs . pruneBy\ cols . pruneBy\ rows \\ &\text{where } pruneBy\ f = f . map\ reduce . f \end{aligned}$$

The paper devoted a great amount of length to the description of the *prune* function. Basically,

- the *reduce* function is a critical component here, which removes the “fixed” options from all of the non-fixed cells of a given list (could be a row, a column, or a box). For example,  $reduce\ [[1], [2, 3], [3]]$  will result in  $[[1], [2], [3]]$ ;
- *pruneBy f* such as *pruneBy rows* will take a *Matrix* of *Choices*, extract all of the rows first, for each row apply the *reduce* function (this is the *map reduce* part), and then wrap it back to a *Matrix* of *Choices*;
- the *prune* function basically repeats this process 3 time for *rows*, *cols*, and *boxs*;
- the critical property of these 3 functions mentioned in the last section is play an important role in this process.

This is now much closer to what we as human beings would do! For example, for row A, because 1, 2, 3, and 8 already exist, the program would prune all these “fixed” numbers from the other cells of this row, and they will all be filled with a *Choices*: [4, 5, 6, 7, 9], which is much better than [1, 2 .. 9]. They will be further pruned at the column and box level.

### 3.3 Further improvement

What have we done so far? Essentially, we generate all kinds of boards, and we apply the rules to check whether the rules can be satisfied. If not, we will drop them. In the “initial solution” part, we leave that “rule check” or “quality control” section to the very end. We produce tons of defective products which will be dropped at the very end, and most of them fail the “quality control” at that point. In the “pruning the choices” version, we made great improvement by removing some of the choices that would obviously fail at the end. We don't allow them to even reach the “matrix cartesian product” part. But all of the *Matrix*s of *Choices* that do reach the “matrix cartesian product” section will

still generate a huge number of boards that are later sent to the “quality control” part. Can we do better here?

If we think about the “matrix cartesian product” strategy, it doesn’t sound efficient at all. Let’s say we have 3 cells in a row and all of them could take one number from 1, 2, and 3 (a choice of [1, 2, 3]). That is to say, we have a list [[1,2,3], [1,2,3], [1,2,3]]. The cartesian products will contain all 27 possible combinations: 111, 112, 113, 121, 122, 123, 131, 132, 133,... 333! We know that there are only 6 valid combinations: 123, 132, 213, 231, 312, 321, because we cannot have duplicates. How do we get this? Well, we look at the first cell and say let’s try 3 rounds for the 3 possible numbers, in the first round we fix it as 1, then we prune the other 2 cells to be [2, 3]; then we fix the second cell, ...; then we go back to the first cell and fix it as 2 for the 2nd round, and so on...

Which cell should we start with then? Well, let’s look back to the section 2 where I described how we as human beings would approach the sudoku problem. We start from the easiest part, which is, the cell(s) that are not fixed yet, but have the smallest flexibility. Once we fix one cell, and do a “prune” that could potentially fix other cells or at least reduce the choices for other cells.

This is exactly what the paper focused on in the final version of the solution. It refined the *filter correct . mcp* part of the function and the final solution becomes:

```
sudoku = extract . search . prune . choices
where extract = map (map (map head))
and search cm
    | blocked cm = []
    | all (all single) cm = [cm]
    | otherwise =
      (concat . map (search . prune) . expand) cm
```

The key function here is the *search* function, when getting a *Matrix* of *Choices* from after pruning,

- If this *Matrix* of *Choices* is “blocked”, which means it either contains some empty cell (for example, after reducing [[1], [1,2], [2]] the middle cell will become empty because there is no choice left) or contains duplicated fixed cells in a row, column or box (for example, after reducing [[1], [1,2], [1,2]]

we will get [[1], [2], [2]], which contains duplicated fixed cells), then *search* will return an empty list, because there is no need to continue;

- If the whole *Matrix* only contains single value cells, this is (almost) a valid sudoku board, we wrap it in a list;
- We can see the previous conditions are base conditions and otherwise, we will do a recursion here: First, the *expand* function scans through the *Choices Matrix*, locates the first cell with the smallest number of choices (let’s say this number is *n*), and expands the original *Choices Matrix* into *n* of *Choices Matrices*, each of which only contains a fixed number at that cell. Then, for each one of the *n Choices Matrices*, *prune* first, then *search* again, and concatenate all of the resulting *Choices Matrices*, which are (almost) valid sudoku boards.

At the end, there is a simple *extract* function (note that the original *map (map head)* step in the paper seems to be wrong because one more layer of *map* is needed) to extract all of values from the single value lists in all cells (for example, [[[[1],[2]], [[3],[4]]]] will become [[[1,2], [3,4]]]), and finally, we get a list of all valid sudoku boards.

In summary, in this final version, we take the *Choices Matrix*, start from the cell with the smallest unfixed cell, and recursively expand the *Matrix* into multiple *Matrices*. In different routes, we repeatedly prune and expand, until we either stop that route as early as we find out that route is not going the right direction, or we get a valid board. This is super smart in that, if we are to fail, then fail early! This can save many of useless attempts and greatly improve the efficiency!

## 4 Discussion

I think this sudoku solver is a very classical example of using Haskell programming to solve problems, and a great example to learn the so-called “wholemeal programming” and the application of equational reasoning in Haskell.

Wholemeal programming means to think big, focus on the whole data structure, instead of the detailed components of the structure (Hinze, 2009). For example, in this sudoku problem, we consider the whole board (the *Matrix*), and most of functions

directly take the whole board as their parameter, instead of digging into the cells. The three helper functions *rows*, *cols*, and *boxes* are the key that make this possible. We can easily extract the cells on all of the three levels (don't need to worry about which exact row or column a cell is in), process them, and then wrap them back into the *Matrix*. How beautiful it is!

Equational reasoning is actually the theme throughout the paper. In the process from the initial solution to the 2nd version, and from the 2nd version to the final version of solution, the authors used equational reasoning to prove that the newer version will lead to the same results as the older version of the sudoku function, based on a series of laws and properties. Some examples of these laws include:

$$\begin{aligned} \text{filter } (p \cdot f) &= \text{map } f \cdot \text{filter } p \cdot \text{map } f \\ \text{filter } (\text{all } p) \cdot cp &= cp \cdot \text{map } (\text{filter } p) \\ \text{filter } \text{nodups} \cdot cp &= \text{filter } \text{nodups} \cdot cp \cdot \text{reduce} \\ \text{map } f \cdot mcp &= mcp \cdot f \text{ if } f \in \{\text{rows}, \text{cols}, \text{boxes}\} \end{aligned}$$

Although I have skipped most of them for the simplicity of this review, I would like to briefly dis-

cuss it here. Just like the basic algebraic properties of numbers, such as commutativity, left or right distributivity, associativity in mathematics, we can use very similar process in functional programming (Hutton, 2007). Equational reasoning is based on the referential transparency, meaning “if you have two expressions that evaluate to be the same thing then you can use one for the other without changing the meaning of the whole program”, according to our course materials(Beckman, Summer 2022). It is not only the basics of equality proof, but also very useful in refactoring the code to make it clear and clean.

## References

- Mattox Beckman. Summer 2022. Progrmg languages compilers. (UIUC CS 421).
- Richard Bird. 2006. Functional pearl: A program to solve sudoku. *Journal of functional programming*, 16(6):671–679.
- Ralf Hinze. 2009. Functional pearl: la tour d’hanoi. *ACM Sigplan Notices*, 44(9):3–10.
- Graham Hutton. 2007. *Programming in Haskell*. Cambridge University Press, USA.