COMP1100: Assignment 3 Report

Student Name: Tien Vu Hoang
Uni ID: U5846163

Student name: Tien Vu Hoang | Student ID: u5846163
Course: COMP1100 | Date: 05/05/2017

Approach

After examining the assignment instruction, I identified 5 main tasks that I needed to perform:

- Create functions to check if a Sudoku is in the correct format (isSudoku)
- Create toString and fromString function to read the input file and present the solution
- Create rows, columns and boxes functions to extract blocks of cells
- Create a solution using backtracking method
- Create an "improved" function to solve more difficult Sudoku.
- Create test to test each funciton

Step 1: isSudoku function

A Sudoku in its standard format should have 9 rows, 9 columns and each cell contains nothing other than a blank or a number (from 1 to 9). To check for all the conditions, I extract all the rows in Sudoku using the given function `cells`. After that I use `length` function to check for the number of elements in each row and column. To check if each cell contains only blank or a number (from 1 to 9), I use `all` function to check every single cell.

Step 2: toString and fromString

2.1 toString

The main function that is responsible for converting a Sudoku to String is `symbolChar = maybe '.' intToDigit`. This function will run through every cell in the Sudoku by applying `map( map symbolChar) (cells Sudoku)` where cells Sudoku will extract 81 cells of the Sudoku.

What `maybe '.' intToDigit` does is that if the value inside a cell in `Nothing`, it will return default value, which is a dot '.'. Otherwise, if the value inside a cell is a Just (1 to 9), function `intToDigit` will be applied to that cell and convert that number into a character.

2.2 fromString

`convert` is the main mechanism behind `fromString`. It is a recursive function that run through 81 characters in the given string. If the character is a dot, '.' or a zero ('0'), the function will return nothing. Otherwise, it converts numeric character into integer type and add the Just to all the number converted.

`listRows` and `splitby9` are created to break the list of cells with 81 character into 9 groups of 9 characters. It allows program to recognize which row a cell belongs to and is indispensable for the solve function. The mechanism behind `splitby9` is recursion. This function will take out the first n elements off the list then continue doing so until the end of the list.

Student name: Tien Vu Hoang
Course: COMP1100

Student ID: u5846163
Date: 05/05/2017

Step 3: rows, columns and boxs

At first, I followed the given instruction by crafting `rows`, `cols` and `boxs` function with the given type `Matrix a -> [Block a]`. Although the functions created with the given instruction work, I decided to change the function type because it allows me to craft better and clearer functions.

As such my `cols` and `rows` functions have the type as: Int -> Sudoku -> Block, while `boxs` function has the type of (Int, Int) -> Sudoku -> Block.

For both `cols` and `rows`, specific row or column are located using (!!) operator. Moreover, to extract columns from a list of rows of cells, I use `transpose` function.

`Boxs` is the most complicated function in this part of the assignment as I have to apply 3 built-in functions `take`, `drop` and `concat` to extract the 9 3x3 Sudoku boxes in each Sudoku. The function for `boxs` is:

```
boxs (x,y) s =
      concat
      $ [take 3 (drop (x*3) rows) | rows <- take 3 (drop (y*3) (cells s))]
```

The first part of the function is `rows <- take 3 (drop (y*3) (cells s))` which assigns the rows that each box includes. For instance, if y = 1, `drop (y*3)` will drop the first three rows in the list of rows. After that `take 3` will select the first three rows in the remaining list, which consists of row 4 to row 9 and dispose the last three rows. As such, in the second part of the function, `take 3 (drop (x*3) rows)` only row 4,5 and 6 undergo `take 3 (drop (x*3)`. If x = 1, row 4,5 and 6 will drop the first three cell of each row. Similarly, `take 3` will select the first three cells of each row and dispose the last three cells. Hence, for x = 1 & y = 1, only cells that belong to rows 4 to 6 and columns 4 to 6 will be selected.

`things_to_check` is a function that combine all the `rows`, `cols` and `boxs` together. This is just a summation of 3 lists of blocks (rows, columns and boxs). This function also specifies the range of (x,y) in `boxs` function which is [0,1,2] (if x = y =3, it will be out of range)

`okBlock` is created to determine if an integer appears twice in a block (row/ column or box). I apply a recursive function to run through every cell in the block. `notElem` helps to assure that no number should appears twice. Finally, `okSudoku` is created to check if all the blocks in the Sudoku does not include any integer twice. The function is implemented with the help of list comprehension `[okBlock b | b <- (things_to_check s)]`. Basically, this run `okBlock` on every block in `things_to_check` list. The “and function returns conjunction of a boolean list. `okSudoku` can only be true if all the block passes the test

Student name: Tien Vu Hoang
Course: COMP1100

Student ID: u5846163
Date: 05/05/2017

Step 4: Solve Sudoku by backtracking

4.1 blank function

`blank` function comprises of `(!!!)` and `blank’`. The purpose of `(!!!)` is to locate the position of a cell in a Sudoku using Haskell operator `(!!)`. Basically, the input of `(!!!)` function is a list of cells and a position, which includes 2 integers. The first integer locates the row while the second locates the column using `(!!)`

`blank’` finds all the cells which contain “Nothing”. The main mechanism behind `blank’` is list comprehension. Function `blank’` will run through every cell in the Sudoku and return position of cells which contain “Nothing”

`blank` function simply extracts the first cell that contain “Nothing” from the list of blank cells by using the built-in function `head`

4.2 (!!=)

`(!!=)` aims to update a tuple with new value at a given index. If the index is invalid (less than 0 or greater than length of list), the list remains unchanged. If the index is within the range (i.e. more than 0 less than length of list minus 1), this function use build-in `take` and `drop` function to inject the value into given position.

```
(take num list) ++ [v] ++ (drop (num+1) list)
```

Given that `num` is the index and v is the value, `take num list` will take out the first `num` characters, which have index ranging from 0 to `num-1`. Then value ‘v’ will be added to the list. `drop (num+1) list` will return character from index `num+1` to the end of the list. As such, element at index `num` is replaced by “v”.

4.3 update

`update` shares the same mechanism with `(!!=)`. The first step that function `update` does is to determine which row the updating takes place using local function `row = (cells s) !! y`.

After that, `take` and `drop` are utilized under the same mechanism as `(!!=)`. However, the value added is determined using `(!!=)`.

```
[row !!= (x, val)]
```

The function above replace current value at index “x” by value “val” at the row that is indicated by “y” in local function `row`

4.4 solve, convert_it, solve’ & solve_it

These four functions are the main components in the crafting of backtracking solution.

`solve_it` consists of three parts, `blank_cell`, `pickanumber` and `s` (input Sudoku). First off, `solve_it` will consider the condition of the given Sudoku. If the given Sudoku is not in the right format, i.e. `not (okSudoku s)`, `solve_it` returns Nothing. In the second case, if there is no blank available and all the cells in the Sudoku are filled, `solve_it` returns Just the Sudoku. Otherwise, `solve_it` returns

```
listToMaybe solutions
where
  solutions = [ fromJust sol | n <- [1..9],
      let sol = solve_it blank_cell pickanumber (update s (blank_cell s) (Just n)),
                      sol /= Nothing]
```

The local `solutions` function is the main mechanism for solving Sudoku using backtracking. Basically, it returns `fromJust sol`, where `sol` can be achieved by updating `blank_cell` with number from 1 to 9. The correct `sol` is achieved when the updated Sudoku does not violate any condition stated in `okSudoku` function.

`listToMaybe` converts `solutions` from `[Sudoku]` type to `Maybe Sudoku` type.

`solve'` assigns `solve_it` to the first blank cell in list of blank cells.

```
solve' = solve_it blank (\s -> Nothing)
```

`solve'` will keep running until there is no more blank cell (or blank return nothing)

`convert_it` and `solve` do the job of transforming the input type from `Sudoku` to `String` and output type from `Maybe Sudoku` to `[String]`. They are not necessary for the program to solve the function. However, they are needed so that intelliJ can compile both Sudoku.hs and Main.hs. Moreover, the type of solve function `solve :: String -> [String]` is also necessary to follow in order for the program to read the .txt files in examples folder.

Step 5: Advanced solution

In the second solution, most of the functions remain unchanged. `solve_it` is still the mechanism used to update the Sudoku. However, the blank function to determine which blank cell is solved first is modified.

Instead of picking the first blank cell using head function, I create a new function called `optimised_blanks`. The logic behind `optimised_blanks` is simple, I calculate the number of blanks in each row, box and column each blank cell locates. The blank cell that locates in row, column and box with least number of blanks should be the easiest blank to solve.

To calculate number of blank in each row, a function named `number_of_blanks` was created. It will take a block (row, column or box), filter out a list of Nothing value and count the number of times Nothing occurs.

Then, `number_of_blanks` function is applied to find number of blanks in row, column and box that contain a specific blank cell. The total number of blank is then added up to form a score for each cell. The position of blank cell is used to computed the score, since it can be used to extract the row, column and box that a blank cell locates.

A list of scores and position for each blank cell is compiled. snd . minimum helps to extract the position of the blank cell with lowest score.

After that, `optimised_blanks` function is used in `solveX = solve_it optimised_blanks (\s -> Nothing)`. The mechanism is similar to that of solve', however, the order of blanks that `solve_it` solves is different as blank is replaced by `optimised_blanks`.

Step 6: Test

The first function that I want to check is `things_to_check`, which helps to compile all the rows, columns and boxes in each Sudoku. This function is important since it is indispensable in `okSudoku` function, which checks if a Sudoku is completely and correctly solved.

To test if `things_to_check` function properly, I want to check whether it always returns a tuple contains 27 lists (9 rows, 9 columns and 9 boxes) and each list contains 9 elements.

The second function that needs to be tested is `okSudoku`. To test for `okSudoku`, I create a local function called `bads`. `bads` checks every single block from `things_to_check` and helps to filter out block that does not meet the requirement specified in `okBlock`. In short, bads returns a list of blocks that violates the conditions.

`prop_okSudoku s = okSudoku s || not (null bads)` specifies the conditions for the test. If all the blocks pass the test, `not (null bads)` will return false and thus, `okSudoku s` must return True to pass the test. Conversely, if there are blocks that fail the `okBlock` test, `not (null bads)` will return True while `okSudoku s` will return False.

The next test I would like to discuss is `prop_updatelist`, which check the function `(!!=)`. The test aim to check whether the total number of elements in the list is unchanged after updating.

The most important test in the program is test for `solve'`. To craft the test, two functions `isAnswer` and `isTheSameSudoku` are created. `isTheSameSudoku` check if two Sudoku are the same by comparing them cell by cell. `isAnswer` check if a Sudoku is the answer of another. If this is the case, the two Sudoku must be the same and must be filled.

`prop_solve'` check if the solution produced by `solve'` is correct by comparing the given Sudoku with answer provided by `solve'`.

Suggestion for improvement

The main reason that urges me to implement `optimised_blanks` is that the backtracking method cannot solve hard.txt effectively and takes more than 5 minutes per puzzle. With the implementation of `optimised_blanks`, the program manages to solve the all the puzzles within hard.txt within 25 minutes or less.

Nonetheless, I believe that there are better solutions that can be applied to enhance the efficiency of the program.

One of the possible improvements is to determine which values can be ignored before applying `solve_it`. Currently, `solve_it` tests all the cases where a blank can be replaced by any number from 1 to 9. However, it is not necessary. We can compute a list of `already_existed` which contains number that appears in all the blocks that a blank belongs to. Then in the list comprehension in `solutions`, values of n should be drawn from [1..9] - `already_existed list` instead of 1 to 9.