Sudoku Solving in Haskell

Robin Martinot, 10589155

Friday 29th June, 2018

Abstract

The goal of this project was to adapt and extend existing code for solving Sudoku puzzles in Haskell to code that solves and generates various different types of Sudokus. Additionally, the difficulty of solving these types of Sudokus was measured by implementing a counter in a recursive function of the code. It seems that with the current measure of complexity it is hard to conclude exactly how 'difficult' Haskell finds the Sudoku, although the new types of Sudokus seemed harder to solve than the regular Sudoku. Nevertheless, the results were rather variable and additional research is required in order to draw any reliable conclusions.

1 Introduction

Implementation of numeric puzzles, in particular Sudokus, can be carried out in many programming languages. Haskell, too, has been used to generate and solve the logic-based number-placement Sudoku puzzle (for existing code, see https://homepages.cwi.nl/ jve/courses/16/fsa/lab/FSAlab3.html). Briefly, the objective of the Sudoku is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid contain all of the digits from 1 to 9. Translating this to code, this entails that there has to be a surjective and injective function $f : \{1, ..., 9\} \rightarrow R$ where R is one of the rows, one of the columns or one of the subgrids.

It is unclear whether code adapted to variations of this puzzle changes the complexity of finding a solution such that it is detectable in Haskell. We might expect that adding extra requirements to the puzzle forces Haskell to check for consistency more often, so that it is less easy to find suitable values. Alternatively, additional requirements might limit the possibilities for new values in open positions, so that it is in fact easier for Haskell to extend the Sudoku with new values.

To test these hypotheses, code was generated in the current project for two different types of Sudokus besides the regular one. A 'diagonal' Sudoku required both diagonals to contain all of the values 1 up to 9. Additionally, a Sudoku type was implemented with four additional 3×3 blocks with left-top corners $(2,2)$, $(2,6)$, $(6,2)$, and $(6,6)$ (coordinates denoting positions by (row, column)). An attempt at comparing the complexity of the different types of Sudokus was made.

2 Methods

(The complete code can be found at the bottom of this report). First, the code for regular Sudokus was adapted to 'diagonal' Sudokus and Sudokus with additional subgrids. This entailed adding a definition of the required additional blocks, and adding a 'Restriction' type which specifies the elements of both diagonals and the additional blocks (besides row, column and subgrid). Instead of writing code for each restriction separately, the Restriction type was added as input argument to functions like 'injectiveFor', 'consistent' and 'freeAtPos', so that these only needed one instance. For example, the latter function, which computes which values are available at a certain position in the grid, now looks like this:

```
freeAtPos :: [Restriction] -> Sudoku -> (Row,Column) -> [Value]
freeAtPos x s (r,c) = foldr1 intersect [ freeInSeq (allValues res s (r,c)) | res
     <- x ]
```

The following lists of restrictions were defined, which describe all of the requirements that each type of Sudoku should adhere to:

```
normal :: [Restriction]
normal = [row, column, subGrid]

diagonal :: [Restriction]
diagonal = [row, column, subGrid, diagonalUp, diagonalDown]

addGrid :: [Restriction]
addGrid = [row, column, subGrid, addsubGrid]
```

In particular, the solveAndShow function now works by taking into account the type of Sudoku:

```
solveAndShow :: Grid -> [Restriction] -> IO ()
solveAndShow gr x = solveShowNs x (initNode gr x)
```

Second, the random Sudoku generator was adapted so that it generates each type of Sudoku. For this, an additional input argument for the list of restrictions that decides the type of Sudoku was added to the existing functions. This way, the code generates a filled Sudoku of the right type and subsequently removes values until no value can be removed without the Sudoku becoming unsolvable.
The generated problem Sudoku is then fed back into the Sudoku solver. For this, the function 'genAndSolve' was defined:

```
genAndSolve :: [Restriction] -> IO ()
genAndSolve x = do (_,s) <- sudGen x
                   showSudoku s
                   solveAndShow (sud2grid s) x
```

Third, to gain more insight into the search process, a counter was added to the recursive function ‘search’. This way, the number of tries Haskell needs to find a solution was kept track of. The function ‘solveAndShow’ returns the number of counted steps along with the solved Sudoku.

```
search :: Int -> (node -> [node])
        -> (node -> Bool) -> [node] -> ([node], Int)
search w _ _ [] = ([],w)
search w children goal (x:xs)
  | goal x    = (x : fst (search w children goal xs), w)
  | otherwise = search (w+1) children goal (children x ++ xs)
```

Finally, the ‘genAndSolve’ function was run 20 times for each type of Sudoku.

```
genAndSolve :: [Restriction] -> IO ()
genAndSolve x = do (_,s) <- sudGen x
                  showSudoku s
                  solveAndShow (sud2grid s) x
```

3 Results

The results are displayed in Table 1, 2 and 3 below. Each column shows the results of running genAndSolve ten times. The average number of steps per column can be found in bold at the bottom, below which the overall average is shown.

Number of steps regular Sudoku	
62	248
133	76
59	594
248	290
232	75
319	473
55	57
59	60
60	96
419	976
164.6	**294.5**
229.55	

Table1. This shows the number of steps Haskell needed to solve the regular Sudoku 20 times.

Number of steps addGrid Sudoku	
238	30343
1598	355681
839	3279
1544	770
1548	13369
938	17853
496	9234
1247	7838
23577	1452
1604	7444
3362.9	**44726.3**
24044.6	

Table2. This shows the number of steps Haskell needed to solve the Sudoku with additional grids 20 times.

Number of steps diagonal Sudoku	
290	264
4376	146
343	1053
1311	1105
9870	26144
548	1926
977	1455
182	220
385	4136
2295	928
2057.7	**3737.7**
2897.7	

Table3. This shows the number of steps Haskell needed to solve the diagonal Sudoku 20 times.

We see that the regular Sudoku on average requires the least number of steps (229.55 steps). Next in line is the diagonal Sudoku with 2897.7 steps on average. The Sudoku with four additional subgrids required most steps on average (24044.6 steps). However, there was one extreme result for this type of Sudoku (355681 steps) which considerably affected this average. Without this particular result, the average for this type of Sudoku is 6590.053 steps instead of 24044.6. Even though the new average is still higher than that of the diagonal Sudoku, this shows that the results are subject to this type of fluctuations.

4 Discussion & conclusion

In short, it appears that the results are relatively inconsistent, although a pattern can be detected. The regular Sudoku seems to require the least number of steps that Haskell needs, followed by the diagonal Sudoku. The Sudoku with four additional blocks requires by far most steps. In line with these results, the approximate computation time for Haskell for diagonal Sudokus and Sudokus with additional blocks exceeded that of the regular Sudokus.

The results might be explained by the fact that normal Sudokus require less checking for Haskell, as they have the least number of requirements that need to be fulfilled. Furthermore, checking for additional subgrids each overlapped with four normal types of grids, perhaps causing the computation to become more complex. The diagonals, although seemingly similar to additional rows or columns, overlap with three normal subgrids and also need to fulfill $r + c == 10$ and $r == c$ for each of their elements. Apparently adding these constraints makes a difference in complexity when solving the Sudoku. Thus, a preliminary conclusion is that diagonal Sudokus and Sudokus with additional grids are indeed harder to solve for Haskell.

In the current project it was assumed that every randomly generated Sudoku was approximately of the same difficulty. This is not straightforward, however: the random Sudoku generator sometimes may have happened to generate a rather easily solvable Sudoku, and other times a hard one. The number of Sudokus that was currently run (20 of each type) is not enough to take away possible effects caused by this (e.g. if it happened to generate easier Sudokus for the normal type in this experiment). In fact, as mentioned in the results section this seems to have been of influence on the averages. Thus, the results cannot yet be taken as reliable.

It should also be kept in mind that there is no obvious and official way to measure the complexity of a Sudoku. The current method was only one way of analyzing the differences between the solving of different types of Sudokus. To give a more complete view of how Haskell solves different Sudokus, measures of other functions, computation time measurements and visualization of intermediate steps in the solving process can be taken into account. To be more precise it is necessary for there to be specific rules for what we define an easy or hard Sudoku problem.

Something I did not have time to complete in the current project is to compare the results to binary puzzles. These are Sudoku-like puzzles that only work with values $0, 1$ and require slightly different rules. Additionally, there are many other types of Sudokus that can still be investigated in Haskell, as well as the mentioned different measures of complexity.

Below, the entire code of the project is displayed, with comments explaining what changes to the existing code were made.

Code Sudoku Solving

The following code integrates the code for solving and generating Sudokus for different types of Sudokus: the usual type, a type with four additional 3x3 blocks with left-top corners (2,2), (2,6), (6,2), and (6,6), and a type where the diagonals require a one-to-one function from 1-9.

```
module Sudoku where
```

```
import Data.List
import System.Random
```

Defining a Sudoku

The following code defines the basic types that are needed to compose a Sudoku and implements a function that prints a Sudoku the way we know it.

```
type Row    = Int
type Column = Int
type Value  = Int
type Grid   = [[Value]]

positions, values :: [Int]
positions = [1..9]
values    = [1..9]

blocks :: [[Int]]
blocks = [[1..3],[4..6],[7..9]]
```

Note that we let 0 represent a blank slot:

```
showVal :: Value -> String
showVal 0 = " "
showVal d = show d
```

```
showRow :: [Value] -> IO()
showRow [a1,a2,a3,a4,a5,a6,a7,a8,a9] = putStrLn . concat $
   [ "| ", showVal a1, " ", showVal a2, " ", showVal a3, " "
   , "| ", showVal a4, " ", showVal a5, " ", showVal a6, " "
   , "| ", showVal a7, " ", showVal a8, " ", showVal a9, " |" ]
showRow _ = error "invalid row"
```

```
showGrid :: Grid -> IO ()
showGrid [as,bs,cs,ds,es,fs,gs,hs,is] =
 do putStrLn "+-------+-------+-------+"
    showRow as; showRow bs; showRow cs
    putStrLn "+-------+-------+-------+"
    showRow ds; showRow es; showRow fs
    putStrLn "+-------+-------+-------+"
    showRow gs; showRow hs; showRow is
    putStrLn "+-------+-------+-------+"
showGrid _ = error "invalid grid"
```

```
type Sudoku = (Row,Column) -> Value
```

```haskell
sud2grid :: Sudoku -> Grid
sud2grid s =
  [ [ s (r,c) | c <- [1..9] ] | r <- [1..9] ]

grid2sud :: Grid -> Sudoku
grid2sud gr = \ (r,c) -> pos gr (r,c)
  where
  pos :: [[a]] -> (Row,Column) -> a
  pos gri (r,c) = (gri !! (r-1)) !! (c-1)
```

```haskell
showSudoku :: Sudoku -> IO ()
showSudoku = showGrid . sud2grid
```

```haskell
bl :: Int -> [Int]
bl x = concat $ filter (elem x) blocks
```

Sudoku with additional grids

For the Sudoku with the four extra 3×3 blocks, we define additional blocks:

```haskell
addblocks :: [[Int]]
addblocks = [[2..4],[6..8]]
```

```haskell
addbl :: Int -> [Int]
addbl x = concat $ filter (elem x) addblocks
```

Restrictions

We now define restrictions that distinguish between different parts of the Sudoku that require injectivity. 'Restriction' gives all coordinates of elements of a certain restriction.

```haskell
type Restriction = (Row,Column) -> [(Row,Column)]
```

allValues gives you list of all values in a certain restriction.

```haskell
allValues :: Restriction -> Sudoku -> (Row, Column) -> [Value]
allValues res s (r,c) = map s (res (r,c))
```

```haskell
row :: Restriction
row (r,_) = [(r, c') | c' <- positions]
```

```haskell
column :: Restriction
column (_, c) = [ (r', c) | r' <- positions]
```

```haskell
subGrid :: Restriction
subGrid (r, c) = [ (r', c') | r' <- bl r, c' <- bl c]
```

```haskell
diagonalUp :: Restriction
diagonalUp (r, c) = [ (r',c') | r' <- positions, c' <- positions, (r' + c') ==
    10, (r + c) == 10 ]
```

```
diagonalDown :: Restriction
diagonalDown (r,c) = [ (r',c') | r' <- positions, c' <- positions, r' == c', r
    == c]
```

```
addsubGrid :: Restriction
addsubGrid (r,c) = [ (r', c') | r' <- addbl r, c' <- addbl c ]
```

```
allRestrictions :: [Restriction]
allRestrictions = [row, column, subGrid, diagonalUp, diagonalDown, addsubGrid]
```

Free values are available values at open slot positions. First we define the general free value function:

```
freeInSeq :: [Value] -> [Value]
freeInSeq sequ = values \\ sequ
```

Then we apply freeInSeq to all the different restrictions that we defined before, and intersect the result. This provides all the free values at a certain position in the Sudoku.

```
freeAtPos :: [Restriction] -> Sudoku -> (Row,Column) -> [Value]
freeAtPos x s (r,c) = foldr1 intersect [ freeInSeq (allValues res s (r,c)) | res
     <- x ]
```

The injectiveFor function now also takes into account the type of Sudoku.

```
injective :: Eq a => [a] -> Bool
injective xs = nub xs == xs
```

```
injectiveFor :: Restriction -> Sudoku -> (Row, Column) -> Bool
injectiveFor x s (r,c) = injective vs where
   vs = filter (/= 0) (map s $ x (r, c)) -- TODO think about this!
```

The consistent function checks for injectivity for every restriction:

```
consistent :: [Restriction] -> Sudoku -> Bool
consistent x s =
   and [ injectiveFor res s (r,c) | res <- x, r <- positions, c <- positions]
```

Solving the Sudoku

The code below implements the checks defined above by adding valid values to the Sudoku problem.

```
extend :: Sudoku -> ((Row,Column),Value) -> Sudoku
extend = update
```

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b
update f (y,z) x = if x == y then z else f x
```

```
type Constraint = (Row,Column,[Value])
type Node = (Sudoku,[Constraint])

showNode :: Node -> IO ()
showNode = showSudoku . fst
```

```
solved  :: Node -> Bool
solved = null . snd
```

```
extendNode :: [Restriction] -> Node -> Constraint -> [Node]
extendNode us (s,constr) (r,c,vs) =
   [(extend s ((r,c),v),
     sortBy length3rd $
         prune us (r,c,v) constr) | v <- vs ]
```

```
length3rd :: (a,b,[c]) -> (a,b,[c]) -> Ordering
length3rd (_,_,zs) (_,_,zs') = compare (length zs) (length zs')
```

```
prune :: [Restriction] -> (Row,Column,Value)
      -> [Constraint] -> [Constraint]
prune _  _ [] = []
prune us (r,c,v) ((x,y,zs):rest)
  | r == x = (x,y,zs\\[v]) : prune us (r,c,v) rest
  | c == y = (x,y,zs\\[v]) : prune us (r,c,v) rest
  | same us (r,c) (x,y) =
        (x,y,zs\\[v]) : prune us (r,c,v) rest
  | otherwise = (x,y,zs) : prune us (r,c,v) rest
```

The sameblock function has been replaced by 'same', which applies the general sameFor function to every type of restriction that it needs. It returns True whenever two positions are in the same (additional) subgrid or diagonal. The condition that it is nonempty is needed so that it will not return True when two elements are e.g. both not in any diagonal.

```
sameFor :: Restriction -> (Row,Column) -> (Row,Column) -> Bool
sameFor u (r,c) (x,y) = (u (r,c) == u (x,y)) && not ( null (u (r, c)))
```

```
same :: [Restriction] -> (Row,Column) -> (Row,Column) -> Bool
same us (r,c) (x,y) = or [ sameFor u (r,c) (x,y) | u <- us ]
```

```
initNode :: Grid -> [Restriction] -> [Node]
initNode gr x = let s = grid2sud gr in
              if not (consistent x s) then []
              else [(s, constraints s x)]
```

```
openPositions :: Sudoku -> [(Row,Column)]
openPositions s = [ (r,c) | r <- positions,
                            c <- positions,
                            s (r,c) == 0 ]
```

```
constraints :: Sudoku -> [Restriction] -> [Constraint]
constraints s x = sortBy length3rd
    [(r,c, freeAtPos x s (r,c)) |
                       (r,c) <- openPositions s ]
```

The search function has been updated with a counter for the number of times that a successor node is tried out. The solveAndShow and genAndSolve function output this number.

```
search :: Int -> (node -> [node])
       -> (node -> Bool) -> [node] -> ([node], Int)
search w _ _ [] = ([],w)
search w children goal (x:xs)
  | goal x     = (x : fst (search w children goal xs), w)
  | otherwise = search (w+1) children goal (children x ++ xs)
```

```
solveNs :: [Restriction] -> [Node] -> ([Node], Int)
solveNs us = search 0 (succNode us) solved

succNode :: [Restriction] -> Node -> [Node]
succNode _ ( _ , [] ) = []
succNode us (s,p:ps) = extendNode us (s,ps) p
```

solveAndShow naturally now also takes into account the type of Sudoku.

```
solveAndShow :: Grid -> [Restriction] -> IO ()
solveAndShow gr x = solveShowNs x (initNode gr x)
```

```
solveShowNs :: [Restriction] -> [Node] -> IO ()
solveShowNs us = (\(ns,w) -> traverse showNode ns >> print [w]) . solveNs us
```

We define the following lists of restrictions to facilitate calling the solve function:

```
normal :: [Restriction]
normal = [row, column, subGrid]
```

```
diagonal :: [Restriction]
diagonal = [row, column, subGrid, diagonalUp, diagonalDown]
```

```
addGrid :: [Restriction]
addGrid = [row, column, subGrid, addsubGrid]
```

Some predefined Sudoku examples that work with the right restriction set:

Example Sudokus

```
examplenormal :: Grid
examplenormal = [[5,3,0,0,7,0,0,0,0],
                 [6,0,0,1,9,5,0,0,0],
                 [0,9,8,0,0,0,0,6,0],
                 [8,0,0,0,6,0,0,0,3],
                 [4,0,0,8,0,3,0,0,1],
                 [7,0,0,0,2,0,0,0,6],
                 [0,6,0,0,0,0,2,8,0],
                 [0,0,0,4,1,9,0,0,5],
                 [0,0,0,0,8,0,0,7,9]]
```

```
examplediag :: Grid
examplediag = [[0,0,0,0,0,0,0,4,0],
               [4,0,0,9,8,0,0,0,6],
               [0,3,0,0,0,7,9,8,0],
               [0,0,0,0,0,0,0,0,0],
               [0,0,5,7,0,4,1,0,0],
               [0,0,0,0,0,0,0,0,0],
               [0,6,3,8,0,0,0,5,0],
               [8,0,0,0,2,6,0,0,3],
               [0,1,0,0,0,0,0,0,0]]
```

```
exampleadd :: Grid
exampleadd = [[0,0,2,0,0,0,0,0,0],
              [0,0,4,0,8,0,9,0,0],
              [0,0,0,3,0,0,0,0,0],
              [0,0,0,0,0,5,4,0,1],
              [0,0,0,0,0,0,0,0,0],
              [5,0,0,2,0,0,0,0,8],
              [0,0,0,0,0,6,0,7,0],
              [0,5,0,0,0,0,0,0,0],
              [0,0,0,0,3,0,0,1,0]]
```

Random Sudoku generation

Now we provide code that generates random Sudokus of each type. A list of restrictions is added as input argument to many functions to let it take into account the Sudoku type.

```
emptyN :: [Restriction] -> Node
emptyN x = (const 0, constraints (const 0) x)
```

```
getRandomInt :: Int -> IO Int
getRandomInt n = getStdRandom (randomR (0,n))
```

```
getRandomItem :: [a] -> IO [a]
getRandomItem [] = return []
getRandomItem xs = do n <- getRandomInt maxi
                      return [xs !! n]
                   where maxi = length xs - 1
```

```
randomize :: Eq a => [a] -> IO [a]
randomize xs = do y <- getRandomItem xs
                  if null y
                    then return []
                    else do ys <- randomize (xs\\y)
                            return (head y:ys)
```

```
sameLen :: Constraint -> Constraint -> Bool
sameLen (_,_,xs) (_,_,ys) = length xs == length ys
```

```
getRandomCnstr :: [Constraint] -> IO [Constraint]
getRandomCnstr cs = getRandomItem (f cs)
  where f [] = []
        f (x:xs) = takeWhile (sameLen x) (x:xs)
```

```
rsuccNode :: [Restriction] -> Node -> IO [Node]
rsuccNode y (s,cs) = do xs <- getRandomCnstr cs
                        if null xs
                          then return []
                          else return
                            (extendNode y (s,cs\\xs) (head xs))
```

```
rsolveNs :: [Restriction] -> [Node] -> IO [Node]
rsolveNs y ns = rsearch (rsuccNode y) solved (return ns)
```

```
rsearch :: (node -> IO [node])
            -> (node -> Bool) -> IO [node] -> IO [node]
rsearch succe goal ionodes =
  do xs <- ionodes
     if null xs
```

```
        then return []
        else
          if goal (head xs)
            then return [head xs]
            else do ys <- rsearch succe goal (succe (head xs))
                    if (not . null) ys
                       then return [head ys]
                       else if null (tail xs) then return []
                            else
                              rsearch
                                succe goal (return $ tail xs)
```

```
genRandomSudoku :: [Restriction] -> IO Node
genRandomSudoku y = do [r] <- rsolveNs y [emptyN y]
                       return r
```

```
uniqueSol :: [Restriction] -> Node -> Bool
uniqueSol w node = singleton (solveNs w [node]) where
  singleton ([],_) = False
  singleton ([_],_) = True
  singleton (_:_:_,_) = False
```

```
eraseS :: Sudoku -> (Row,Column) -> Sudoku
eraseS s (r,c) (x,y) | (r,c) == (x,y) = 0
                     | otherwise      = s (x,y)
```

```
eraseN :: [Restriction] -> Node -> (Row,Column) -> Node
eraseN x n (r,c) = (s, constraints s x)
  where s = eraseS (fst n) (r,c)
```

```
minimalize :: [Restriction] -> Node -> [(Row,Column)] -> Node
minimalize _ n [] = n
minimalize x n ((r,c):rcs) | uniqueSol x n' = minimalize x n' rcs
                           | otherwise    = minimalize x n  rcs
  where n' = eraseN x n (r,c)
```

```
filledPositions :: Sudoku -> [(Row,Column)]
filledPositions s = [ (r,c) | r <- positions,
                              c <- positions, s (r,c) /= 0 ]
```

```
genProblem :: [Restriction] -> Node -> IO Node
genProblem x n = minimalize x n <$> randomize xs
     where xs = filledPositions (fst n)
```

sudGen generates a filled-in Sudoku of the right type (r), prints it, and subsequently prints the problem version (s).

```
sudGen :: [Restriction] -> IO (Sudoku,Sudoku)
sudGen x = do [r] <- rsolveNs x [emptyN x]
              s  <- genProblem x r
              return (fst r, fst s)
```

Generating and solving a Sudoku problem

genAndSolve shows a random problem Sudoku of the right type, and continues to solve it.

```
genAndSolve :: [Restriction] -> IO ()
genAndSolve x = do (_,s) <- sudGen x
                   showSudoku s
                   solveAndShow (sud2grid s) x
```

To run the genAndSolve function multiple times the function below can be used, or alternatively the replicateM function does the same thing.

```
times :: Int -> [Restriction] -> IO ()
times 0 _ = return ()
times n x = do genAndSolve x
               times (n-1) x
```

References

Code for regular Sudokus provided by Jan van Eijck, https://homepages.cwi.nl/ jve/-courses/16/fsa/lab/FSAlab3.html