A Simple Sudoku Solver
27th September, 2007
In Chapter 05

Explanations are added in Spring 2019 for CIS 623

A. The Sudoku game

Given M, a 9 X 9 matrix of integers, where each cell is either empty (a blank) or a number k where 0 < k < 10, fill in the empty cells with the digits 1 to 9 so that each row, column and 3×3 box contains the numbers 1 to 9.

B. Specification

0. Introduce the basic data types

```
> type Matrix a = [Row a]
> type Row a    = [a]

> type Grid     = Matrix Digit
> type Digit    = Char

> digits  :: [Digit]
> digits  =  ['1'..'9']

> blank   :: Digit -> Bool
> blank   =  (== '0')
```

Remarks:

i. a sudoku `gameboard' is of type [[Char]].
   For example: eg1 is the sudoku grid given
   in Bird, Figure 5.1.

```
> eg1 :: [[Char]]
> eg1 = [
>  ['0','0','4','0','0','5','7','0','0'],
>  ['0','0','0','0','0','9','4','0','0'],
>  ['3','6','0','0','0','0','0','0','8'],
>  ['7','2','0','0','6','0','0','0','0'],
>  ['0','0','0','4','0','2','0','0','0'],
>  ['0','0','0','0','8','0','0','9','3'],
>  ['4','0','0','0','0','0','0','5','6'],
>  ['0','0','5','3','0','0','0','0','0'],
>  ['0','0','6','1','0','0','9','0','0']]
```

Another example, eg2 (solvable), is added below:

```
> eg2 :: [[Char]]
> eg2 = [
>  ['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']]
```

---

1. Specify a solution (brute force)

We begin with a simple brute force solution solve, using two subsidiary functions:

```
solve :: Grid -> [Grid]
solve = filter valid . completions
```

where the function completions will take the given grid as input, complete it by filling in every possible choice for the blank entries, and, the function valid will test, for each of the filled grids, if it contains duplicates in any row, box or column. By filtering out the invalid entries, we will obtain a solution to the given sudoku puzzle.

We can define completions by a two way process:

```
> solve1 :: Grid -> [Grid]
> solve1 = filter valid . expand . choices
```

where choices install the available digits for each cell. For example, choices eg1 will give

```
*Main> choices eg1
[
["123456789","123456789","4","123456789","123456789",
"5","7","123456789","123456789"],
["123456789","123456789","123456789","123456789",
"123456789","9","4","123456789","123456789"],
... 7 more lists (omitted)
]
```

i. Step 1: implementation of the function choices

The definition of the function choices is given below:

```
> type Choices = [Digit]

> choices :: Grid -> Matrix Choices
> choices = map (map choice)
>  where choice d | blank d   = digits
>                 | otherwise = [d]
```

For the function expand, we make use of cartesian product. Recall that, given two sets A and B, the cartesian product of A and B is:

A x B = {(a,b): a is in set A, b is in set B}

For example, if we want to list all possible ways to fill the empty cells by 1,2 or 3 in the matrix eg3 shown below:

```
-------------
| 1 |   |   |
-------------
|   | 2 |   |      <--- matrix eg3
-------------
| 2 |   | 3 |
-------------
```

```
> eg3 :: [[Char]]
> eg3 = [
>  ['1','0','0'],
>  ['0','2','0'],
>  ['2','0','3']
>  ]

> digit123  :: [Digit]
> digit123  =  ['1'..'3']

> newchoices = map (map newchoice)
> newchoice d = if blank d then digit123 else [d]

     we can compute (cp . map cp) (newchoices eg3)
     to list all possible ways (3^5 = 243 ways) to
     fill in the matrix.

     Note: One may like to verify that as follows:

     *main> length ((cp . map cp) (newchoices eg3))
     243

     The function cp (cartesian product) and the
     function expand (uses cp) are implemented as
     follows:

> expand :: Matrix Choices -> [Grid]
> expand = cp . map cp

> cp :: [[a]] -> [[a]]
> cp []       = [[]]
> cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]

     i. Step 2: implementation of the function valid

        Let g be a sudoku grid where the entry at each
        cell is either 1,2, ..,9.  Then g is a valid
        solution to a sudoku game if

        (1) there is no dups at each row of g, and
        (2) there is no dups at each col of g, and
        (3) there is no dups at each box of g

        Hence, we define the function valid as:

> valid  :: Grid -> Bool
> valid g = all nodups (rows g) &&
>           all nodups (cols g) &&
>           all nodups (boxs g)

        We define the function nodups by recursion.
        The function x `notElem` xs checks if an
        element x is a member of xs.

> nodups       :: Eq a => [a] -> Bool
> nodups []     = True
> nodups (x:xs) = x `notElem` xs && nodups xs

       The function rows and cols are defined as
       follows (Note the use of zipWith when
       defining cols):

> rows :: Matrix a -> [Row a]
> rows = id
```

```
> cols          :: Matrix a -> [Row a]
> cols [xs]     = [[x] | x <- xs]
> cols (xs:xss) = zipWith (:) xs (cols xss)
```

The definition of boxs is built
from a sequence of operations.

```
> boxs :: Matrix a -> [Row a]
> boxs = map ungroup . ungroup . map cols .
>          group . map group
```

where

```
> ungroup          = concat
> group []         = []
> group (x:y:z:xs) = [x,y,z]:group xs
```

It is instructive to display the intermediate
results for boxs eg1 to show how the sequence
of functions works.

```
*Main> map group eg1
*Main> (group . map group) eg1
*Main> (map cols . group . map group) eg1
*Main> (ungroup . map cols . group . map group) eg1
*Main>
(map ungroup . ungroup . map cols . group . map group) eg1
```

the results will be shown in a separate document.

To reinforce your understanding of the development
of the solution, prove (see 5.2):

```
rows . rows = id  -- trivial
cols . cols = id  -- exercise
boxs . boxs = id  -- completed proof given in 5.2
```

---

2. Improve the solution by Pruning

Assuming about 20 of the 81 entries in a sudoku grid
are fixed initially, then the number of choices generated
by the function (expand . choices) is 9^{61}. As we need
to avoid checking all of them, we use a pruning strategy:

strategy:

remove any choices from a cell c that already occur as
singleton entries in the row, column and box containing c.

We therefore seek a function

```
prune :: Matrix [Digit] -> Matrix [Digit]
```

so that

```
filter valid . expand = filter valid . expand . prune
```

i. The design of the prune function

We begin by considering a special case where we prune by row. That is, we remove any choices from a cell c that already occur as singleton entries in the row containing c. This is acheived by the function pruneRow (with the help of the subsidiary function remove).

```
> pruneRow :: Row Choices -> Row Choices
> pruneRow row = map (remove ones) row
>  where ones = [d | [d] <- row]

> remove :: Choices -> Choices -> Choices
> remove xs [d] = [d]
> remove xs ds  = filter (`notElem` xs) ds
```

Observe that the cols function (resp. the boxs function) rearranges the columns (resp. boxs) of the input g (sudoku grid) to rows of the output sudoku grid. Hence, the cases where we prune by columns (resp. by boxs) can be carried out in a similar manner. To be precise, the action `prune by rows', `prune by columns' and `prune by boxs' can be carried out by using the pruneBy function specified as

```
pruneBy f = f . map pruneRow . f
```

Hence, our prune function is:

```
> prune :: Matrix Choices -> Matrix Choices
> prune =
>  pruneBy boxs . pruneBy cols . pruneBy rows
>  where pruneBy f = f . map pruneRow . f
```

Remark:

Now, Our solver can be expressed as:

```
solve = ( search ) . choices
```

We will incorporate our prune function as part of our search algorithm.

ii. The verification of the prune function

We can prove, by equation reasoning (section 5.3), that

```
filter valid . expand = filter valid . expand . prune
```

iii. An alternate approach

one way to implement a search algorithm is to apply the prune function until one singleton choices are left. We can use the function many to implement this idea:

```
> many :: (Eq a) => (a -> a) -> a -> a
> many f x = if x==y then x else many f y
>     where y = f x

> solve3 = filter valid . expand . many prune . choices
```

```
        Remark:

        The result of many prune . choices is a matrix
        of choices that can be put into one of three classes:

        1. A complete matrix in which every entry is a
           singleton choice. In this case expand will extract
           a single grid that can be checked for validity.
        2. A matrix that contains the empty choice somewhere.
           In this case expand will produce the empty list.
        3. A matrix that does not contain the empty choice but
           does contain some entry with two or more choices.

           In case 3, rather than carry out full expansion, a more
           sensible idea is to make use of a partial expansion that
           installs the choices for just one of the entries, and to
           start the pruning process again on each result.

3. Single-cell expansion

           The following function expand1 implement a single-cell
           expansion.

> expand1   :: Matrix Choices -> [Matrix Choices]
> expand1 rows =
>  [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
>  where
>  (rows1,row:rows2) = break (any smallest) rows
>  (row1,cs:row2)    = break smallest row
>  smallest cs       = length cs == n
>  n                 = minimum (counts rows)

> counts = filter (/=1) . map length . concat

           To understand how it works, consider using the
           examples eg1 and eg2 and examine:

           expand1 [eg1]
           expand2 [eg2]
           ---

           We leave the details experiments to the reader.

           discussions: how does the function resembles
           `expanding a node in a search tree' ?

4. Final algorithm

           The final algorithm uses the following search
           algorithm:

> solve2 :: Grid -> [Grid]
> solve2 =  search . choices

> search :: Matrix Choices -> [Grid]
> search cm -- cm: the list of possible choices
>  |not (safe pm)  = []  -- return null if dups detected
>  |complete pm    = [map (map head) pm] -- search's done
>  |otherwise      = (concat . map search . expand1) pm
>                    -- expand the search tree one step
>                    -- note the use of 'prune' at each
>                    -- step
>  where pm = prune cm  -- pm: results after pruning cm
```

```
> complete :: Matrix Choices -> Bool
> complete = all (all single) -- test if `search' completes

> single [_] = True
> single _   = False


> safe :: Matrix Choices -> Bool
>         -- test if each grid in the input list
>         -- is `ok' (no duplicates)
> safe cm = all ok (rows cm) &&
>           all ok (cols cm) &&
>           all ok (boxs cm)

> ok row = nodups [d | [d] <- row]
```

5. Exercises

Attempt Exercise A to H and study the given solution the text provides.