

Sudoku Solver

Brian Alliet

October 12, 2005

1 Introduction

This Haskell module implements a solver for Sudoku¹ puzzles. It can solve any Sudoku puzzle, even those that require backtracking.

2 Data Types

```
data CellState a = Known a | Unknown [a] | Impossible deriving Eq
```

Each cell in a Sudoku grid can be in one of three states: “Known” if it has a known correct value², “Unknown” if there is still more than one possible correct value, or “Impossible” if there is no value that can possibly fit the cell. Sudoku grids with “Impossible” cells are quickly discarded by the `solve` function.

```
type Coords = (Int,Int)
type Grid a = Array Coords (CellState a)
newtype Sudoku a = Sudoku { unSudoku :: Grid a } deriving Eq
```

We represent a Sudoku grid as an Array indexed by integer coordinates. We additionally define a newtype wrapper for the grid. The smart constructor, `makeSudoku` verifies some invariants before creating the Sudoku value. All the public API functions operate on the Sudoku type.

```
instance Show a => Show (Sudoku a) where showsPrec p = showParen (p>0) . showsGrid . unSudoku
instance Show a => Show (CellState a) where showsPrec _ = showsCell
```

We define Show instances for the above types.

¹<http://en.wikipedia.org/wiki/Sudoku>

²Actually this doesn't always mean it is correct. While we are in the backtracking stage we make our guesses “Known”.

3 Internal Functions

```
size :: Grid a → Int
size = (+1).fst.snd.bounds
```

size returns the size (the width, height, and number of subboxes) for a Sudoku grid. We ensure Grid's are always square and indexed starting at (0,0) so simply incrementing either of the array's upper bounds is correct.

```
getRow, getCol, getBox :: Grid a → Int → [(Coords, CellState a)]
getRow grid r = [let l = (r,c) in (l,grid!l)|c ← [0..size grid - 1]]
getCol grid c = [let l = (r,c) in (l,grid!l)|r ← [0..size grid - 1]]
getBox grid b = [let l = (r,c) in (l,grid!l)|r ← [boxR..boxR+boxN-1],c ← [boxC..boxC+boxN-1]]
  where
    boxN = intSqrt (size grid); boxR = b `quot` boxN * boxN; boxC = b `rem` boxN * boxN

getBoxOf :: Grid a → Coords → [(Coords, CellState a)]
getBoxOf grid (r,c) = grid `getBox` ((r `quot` boxN * boxN) + (c `quot` boxN))
  where boxN = intSqrt (size grid)
```

getRow, getCol, and getBox return the coordinates and values of the cell in row, column, or box number n, r, or b.

```
getNeighbors :: Eq a ⇒ Grid a → Coords → [(Coords, CellState a)]
getNeighbors grid l@(r,c) = filter ((≠l).fst)
  $ foldr (union.$grid) []
  [(`getRow`r),(`getCol`c),(`getBoxOf`l)]
```

getNeighbors returns the coordinates and values of all the neighbors of this cell.

```
impossible :: Eq a ⇒ Grid a → Coords → [a]
impossible grid l = map snd $ justKnowns $ grid `getNeighbors` l
```

impossible returns a list of impossible values for a given cell. The impossible values consist of the values any "Known" neighbors.

```
justUnknowns :: [(Coords, CellState a)] → [(Coords, [a])]
justUnknowns = foldr (λc → case c of (p,Unknown xs) → ((p,xs):); _ → id) []

justKnowns :: [(Coords, CellState a)] → [(Coords, a)]
justKnowns = foldr (λc → case c of (p,Known x) → ((p,x):); _ → id) []
```

justUnknowns and justKnowns return only the Known or Unknown values (with the constructor stripped off) from a list of cells.

```
updateGrid :: Grid a → [(Coords, CellState a)] → Maybe (Grid a)
updateGrid _ [] = Nothing
updateGrid grid xs = Just $ grid // xs
```

updateGrid applies a set of updates to a grid and returns the new grid only if it was updated.

4 Public API

```
makeSudoku :: (Num a, Ord a, Enum a) => [[a]] -> Sudoku a
makeSudoku xs
  | not (all ((==size).length) xs) = error "error not a square"
  | (intSqrt size)^(2::Int) /= size = error "error dims aren't perfect squares"
  | any (\x -> x < 0 || x > fromIntegral size) (concat xs) = error "value out of range"
  | otherwise = Sudoku (listArray ((0,0),(size-1,size-1)) states)
  where
    size = length xs
    states = map f (concat xs)
    f 0 = Unknown [1..fromIntegral size]
    f x = Known x
```

`makeSudoku` makes a `Sudoku` value from a list of numbers. The given matrix must be square and have dimensions that are a perfect square. The possible values for each cell range from 1 to the dimension of the square with “0” representing unknown values.³

```
eliminate :: Eq a => Sudoku a -> Maybe (Sudoku a)
eliminate (Sudoku grid) = fmap Sudoku $ updateGrid grid changes >>= sanitize
  where
    changes = concatMap findChange $ assocs grid
    findChange (l,Unknown xs)
      = map ((,) l)
        $ case filter (not.('elem' impossible grid l)) xs of
            [] -> return Impossible
            [x] -> return $ Known x
            xs'
              | xs' /= xs -> return $ Unknown xs'
              | otherwise -> mzero
    findChange _ = mzero
    sanitize grid = return $ grid // [(l,Impossible) |
      (l,x) <- justKnowns changes, x 'elem' impossible grid l]
```

The `eliminate` phase tries to remove possible choices for “Unknowns” based on “Known” values in the same row, column, or box as the “Unknown” value. For each cell on the grid we find its “neighbors”, that is, cells in the same row, column, or box. Out of those neighbors we get a list of all the “Known” values. We can eliminate all of these from our list of candidates for this cell. If we’re lucky enough to eliminate all the candidates but one we have a new “Known” value. If we’re unlucky enough to have eliminated **all** the possible candidates we have a new “Impossible” value.

After iterating through every cell we make one more pass looking for conflicting changes. `sanitize` marks cells as “Impossible” if we have conflicting “Known” values.

³The rest of the code doesn’t depend on any of this weird “0” is unknown representation. In fact, it doesn’t depend on numeric values at all. “0” is just used here because it makes representing grids in Haskell source code easier.

```

analyze :: Eq a => Sudoku a -> Maybe (Sudoku a)
analyze (Sudoku grid) = fmap Sudoku $ updateGrid grid $ nub [u |
    f <- map ($grid) [getRow, getCol, getBox],
    n <- [0..size grid - 1],
    u <- unique (f n)]
  where
    unique xs = foldr f [] $ foldr (union.snd) [] unknowns \\ map snd (justKnowns xs)
      where
        unknowns = justUnknowns xs
        f c = case filter ((c'elem').snd) unknowns of
            [(p,_) ] -> ((p, Known c):)
            _ -> id

```

The analyze phase tries to turn “Unknowns” into “Knowns” when a certain “Unknown” is the only cell that contains a value needed in a given row, column, or box. We apply each of the functions `getRow`, `getCol`, and `getBox` to all the indices on the grid, apply `unique` to each group, and update the array with the results. `unique` gets a list of all the unknown cells in the group and finds all the unknown values in each of those cells. Each of these values are iterated though looking for a value that is only contained in one cell. If such a value is found the cell containing it must be that value.

```

backtrack :: (MonadPlus m, Eq a) => Sudoku a -> m (Sudoku a)
backtrack (Sudoku grid) = case (justUnknowns (assocs grid)) of
    [] -> return $ Sudoku grid
    ((p,xs):_) -> msum $ map (\x -> solve $ Sudoku $ grid // [(p, Known x)]) xs

```

Sometimes the above two phases still aren’t enough to solve a puzzle. For these rare puzzles backtracking is required. We attempt to solve the puzzle by replacing the first “Unknown” value with each of the candidate values and solving the resulting puzzles. Hopefully at least one of our choices will result in a solvable puzzle.

We could actually solve any puzzle using backtracking alone, although this would be very inefficient. The above functions simplify most puzzles enough that the backtracking phase has to do hardly any work.

```

solve :: (MonadPlus m, Eq a) => Sudoku a -> m (Sudoku a)
solve sudoku =
  case eliminate sudoku of
    Just new
      | any (==Impossible) (elems (unSudoku new)) -> mzero
      | otherwise -> solve new
    Nothing -> case analyze sudoku of
      Just new -> solve new
      Nothing -> backtrack sudoku

```

`solve` glues all the above phases together. First we run the `eliminate` phase. If that found the puzzle to be unsolvable we abort immediatly. If `eliminate` changed the grid we go though the `eliminate` phase again hoping to eliminate more. Once `eliminate` can do no more work we move on to the `analyze` phase. If this succeeds in doing some work we start over again with the `eliminate` phase. Once `analyze` can do no more work we have no choice but to resort to backtracking. (However in most cases backtracking won’t actually do anything because the puzzle is already solved.)

```

showsCell :: Show a => CellState a -> Shows
showsCell (Known x) = shows x
showsCell (Impossible) = showChar 'X'
showsCell (Unknown xs) = \rest -> ((' ':)
    $ foldr id (' ':rest)
    $ intersperse (showChar ' ')
    $ map shows xs

```

showCell shows a cell.

```

showsGrid :: Show a => Grid a -> Shows
showsGrid grid = showsTable [[grid!(r,c) | c <- [0..size grid-1]] | r <- [0..size grid-1]]

```

showGrid show a grid.

```

— FEATURE: This is pretty inefficient
showsTable :: Show a => [[a]] -> Shows
showsTable xs = (showChar '\n' .) $ showString $ unlines $ map (concat . intersperse " ") xs''
  where
    xs' = (map.map) show xs
    colWidths = map (max 2 . maximum . map length) (transpose xs')
    xs'' = map (zipWith (\n s -> s ++ (replicate (n - length s) ' ')) colWidths) xs'

```

showsTable shows a table (or matrix). Every column has the same width so things line up.

```

intSqrt :: Integral a => a -> a
intSqrt n
  | n < 0 = error "intSqrt: negative n"
  | otherwise = f n
  where
    f x = if y < x then f y else x
          where y = (x + (n `quot` x)) `quot` 2

```

intSqrt is Newton's Iteration for finding integral square roots.