Sudoku

From HaskellWiki

(Redirected from 99 questions/Solutions/97)

Here are a few Sudoku solvers coded up in Haskell...

Contents

1 Monadic non-deterministic solver

Here is a solver by CaleGibbard. It possibly looks even more naïve than it actually is. This does a backtracking search, trying possibilities until it finds one which

works, and backtracking when it can no longer make a legal move.

```
import MonadNondet (option)
import Sudoku
import System
import Control.Monad

solve = forM [(i,j) | i <- [1..9], j <- [1..9]] $ \(i,j) -> do
    v <- valAt (i,j)       -- ^ for each board position
    when (v == 0) $ do     -- if it's empty (we represent that with a 0)
        a <- option [1..9] -- pick a number
        place (i,j) a      -- and try to put it there

main = do
    [f] <- getArgs
    xs <- readFile f
    putStrLn . evalSudoku $ do { readSudoku xs; solve; showSudoku }
```

Now, to the meat of the thing, the monad which makes the above look so nice. We construct a monad which is suitable for maintaining Sudoku grids and trying options nondeterministically. Note that outside of this module, it's impossible to create a state which has an invalid Sudoku grid, since the only way to update the state handles the check to ensure that the move is legal.

```
{-# OPTIONS_GHC -fglasgow-exts #-}
module Sudoku
    (Sudoku,
     readSudoku,
     runSudoku,
     evalSudoku,
     execSudoku,
     showSudoku,
     valAt, rowAt, colAt, boxAt,
     place)
     where
import Data.Array.Diff
import MonadNondet
import Control.Monad.State

-- Nondet here is a drop-in replacement for [] (the list monad) which just runs a little fast
newtype Sudoku a = Sudoku (StateT (DiffUArray (Int,Int) Int) Nondet a)
    deriving (Functor, Monad, MonadPlus)

{- -- That is, we could also use the following, which works exactly the same way.
newtype Sudoku a = Sudoku (StateT (DiffUArray (Int,Int) Int) [] a)
    deriving (Functor, Monad, MonadPlus)
-}

initialSudokuArray = listArray ((1,1),(9,9)) [0,0..]

runSudoku (Sudoku k) = runNondet (runStateT k initialSudokuArray)

evalSudoku = fst . runSudoku
execSudoku = snd . runSudoku

showSudoku = Sudoku $ do
    a <- get
    return $ unlines [unwords [show (a ! (i,j)) | j <- [1..9]] | i <- [1..9]]
```

```
readSudoku :: String -> Sudoku ()
readSudoku xs = sequence_ $ do
    (i,ys) <- zip [1..9] (lines xs)
    (j,n)  <- zip [1..9] (words ys)
    return $ place (i,j) (read n)

valAt' (i,j) = do
    a <- get
    return (a ! (i,j))

rowAt' (i,j) = mapM valAt' [(i, k) | k <- [1..9]]

colAt' (i,j) = mapM valAt' [(k, j) | k <- [1..9]]

boxAt' (i,j) = mapM valAt' [(i' + u, j' + v) | u <- [1..3], v <- [1..3]]
  where i' = ((i-1) `div` 3) * 3
        j' = ((j-1) `div` 3) * 3

valAt = Sudoku . valAt'
rowAt = Sudoku . rowAt'
colAt = Sudoku . colAt'
boxAt = Sudoku . boxAt'

-- This is the least trivial part.
-- It just guards to make sure that the move is legal,
-- and updates the array in the state if it is.
place :: (Int,Int) -> Int -> Sudoku ()
place (i,j) n = Sudoku $ do
    v <- valAt' (i,j)
    when (v == 0 && n /= 0) $ do
        rs <- rowAt' (i,j)
        cs <- colAt' (i,j)
        bs <- boxAt' (i,j)
        guard $ (n `notElem`) $ rs ++ cs ++ bs
        a <- get
        put (a // [((i,j),n)])
```

This is a fast NonDeterminism monad. It's a drop-in replacement for the list monad in this case. It's twice as fast when compiled with optimisations but a little slower without. You can also find it on the wiki at NonDeterminism.

I've made a few small modifications to this one to hopefully make it more concretely readable.

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module MonadNondet where

import Control.Monad
import Control.Monad.Trans

import Control.Monad.Identity

newtype NondetT m a
  = NondetT { foldNondetT :: (forall b. (a -> m b -> m b) -> m b -> m b) }

runNondetT :: (Monad m) => NondetT m a -> m a
runNondetT m = foldNondetT m (\x xs -> return x) (error "No solution found.")
```

```
instance (Functor m) => Functor (NondetT m) where
  fmap f (NondetT g) = NondetT (\cons nil -> g (cons . f) nil)

instance (Monad m) => Monad (NondetT m) where
  return a = NondetT (\cons nil -> cons a nil)
  m >>= k  = NondetT (\cons nil -> foldNondetT m (\x -> foldNondetT (k x) cons) nil)

instance (Monad m) => MonadPlus (NondetT m) where
  mzero         = NondetT (\cons nil -> nil)
  m1 `mplus` m2 = NondetT (\cons -> foldNondetT m1 cons . foldNondetT m2 cons)

instance MonadTrans NondetT where
  lift m = NondetT (\cons nil -> m >>= \a -> cons a nil)

newtype Nondet a = Nondet (NondetT Identity a) deriving (Functor, Monad, MonadPlus)
runNondet (Nondet x) = runIdentity (runNondetT x)

foldNondet :: Nondet a -> (a -> b -> b) -> b -> b
foldNondet (Nondet nd) cons nil =
   runIdentity $ foldNondetT nd (\x xs -> return (cons x (runIdentity xs))) (return nil)

option :: (MonadPlus m) => [a] -> m a
option = msum . map return
```

2 Simple solver

By AlsonKemp. This solver is probably similar to Cale's but I don't grok the non-deterministic monad...

Note: this solver is exhaustive and will output all of the solutions, not just the first one. In order to make it non-exhaustive, add a case statement to solve' in order to check "r" and branch on the result.

```
import System
import Control.Monad
import Data.List
import Data.Array.IO

type SodokuBoard = IOArray Int Int

main = do
    [f] <- getArgs
    a <- newArray (1, 81) 0
    readFile f >>= readSodokuBoard a
    putStrLn "Original:"
    printSodokuBoard a
    putStrLn "Solutions:"
    solve a (1,1)

readSodokuBoard a xs = sequence_ $ do (i,ys) <- zip [1..9] (lines xs)
                                      (j,n)  <- zip [1..9] (words ys)
```

```
                                    return $ writeBoard a (j,i) (read n)

printSodokuBoard a =
   let printLine a y =
     mapM (\x -> readBoard a (x,y)) [1..9] >>= mapM_ (putStr . show) in do
                    putStrLn "-----------"
                    mapM_ (\y -> putStr "|" >> printLine a y >> putStrLn "|") [1..9]
                    putStrLn "-----------"

-- the meat of the program.  Checks the current square.
-- If 0, then get the list of nums and try to "solve' "
-- Otherwise, go to the next square.
solve :: SodokuBoard  -> (Int, Int) -> IO (Maybe SodokuBoard)
solve a (10,y) = solve a (1,y+1)
solve a (_, 10)= printSodokuBoard a >> return (Just a)
solve a (x,y)  = do v <- readBoard a (x,y)
                    case v of
                      0 -> availableNums a (x,y) >>= solve' a (x,y)
                      _ ->  solve a (x+1,y)
     -- solve' handles the backtacking
  where solve' a (x,y) []     = return Nothing
        solve' a (x,y) (v:vs) = do writeBoard a (x,y) v   -- put a guess onto the board
                                   r <- solve a (x+1,y)
                                   writeBoard a (x,y) 0   -- remove the guess from the board
                                   solve' a (x,y) vs      -- recurse over the remainder of th

-- get the "taken" numbers from a row, col or box.
getRowNums a y = sequence [readBoard a (x',y) | x' <- [1..9]]
getColNums a x = sequence [readBoard a (x,y') | y' <- [1..9]]
getBoxNums a (x,y) = sequence [readBoard a (x'+u, y'+v) | u <- [0..2], v <- [0..2]]
  where x' = (3 * ((x-1) `quot` 3)) + 1
        y' = (3 * ((y-1) `quot` 3)) + 1

-- return the numbers that are available for a particular square
availableNums a (x,y) = do r <- getRowNums a y
                           c <- getColNums a x
                           b <- getBoxNums a (x,y)
                           return $ [0..9] \\ (r `union` c `union` b)

-- aliases of read and write array that flatten the index
readBoard a (x,y) = readArray a (x+9*(y-1))
writeBoard a (x,y) e = writeArray a (x+9*(y-1)) e
```

3 Complete decision tree

By Henning Thielemann.

```
module Sudoku where

{-
  This is inspired by John Hughes "Why Functional Programming Matters".
  We build a complete decision tree.
  That is, all alternatives in a certain depth
  have the same number of determined values.
  At the bottom of the tree all possible solutions can be found.
  Actually the algorithm is very stupid:
  In each depth we look for the field with the least admissible choices of numbers
  and prune the alternative branches for the other fields.
-}
```

```haskell
import Data.Char (ord, chr, isDigit, digitToInt, intToDigit)
import Data.Array (Array, range, (!), (//))
import Data.Tree (Tree)
import qualified Data.Tree as Tree
import Data.List (sort, minimumBy)
import Data.Maybe (catMaybes, isNothing, fromMaybe, fromJust)
import qualified Data.Array as Array

{-
Example:

ghci -Wall Sudoku.hs

*Sudoku> mapM_ putCLn (solutions exampleHawiki0)
-}


{- [[ATree]] contains a list of possible alternatives for each position -}
data ATree a = ANode T [[ATree a]]

type Coord   = Int
type Address = (Int,Int,Int,Int)
type Element = Int

type T        = Array Address (Maybe Element)
type Complete = Array Address Element

fieldBounds :: (Address, Address)
fieldBounds = ((0,0,0,0), (2,2,2,2))

squareRange :: [(Coord, Coord)]
squareRange = range ((0,0), (2,2))

alphabet :: [Element]
alphabet = [1..9]



{- * solution -}

{-
  Given two sorted lists,
  remove the elements of the first list from the second one.
-}
deleteSorted :: Ord a => [a] -> [a] -> [a]
deleteSorted [] ys = ys
deleteSorted _  [] = []
deleteSorted (x:xs) (y:ys) =
   case compare x y of
      EQ -> deleteSorted xs ys
      LT -> deleteSorted xs (y:ys)
      GT -> y : deleteSorted (x:xs) ys

admissibleNumbers :: [[Maybe Element]] -> [Element]
admissibleNumbers =
   foldl (flip deleteSorted) alphabet .
   map (sort . catMaybes)

admissibleAdditions :: T -> Address -> [Element]
admissibleAdditions sudoku (i,j,k,l) =
```

```
   admissibleNumbers (map ($ sudoku)
      [selectRow    (i,k),
       selectColumn (j,l),
       selectSquare (i,j)])

allAdmissibleAdditions :: T -> [(Address, [Element])]
allAdmissibleAdditions sudoku =
   let adds addr =
          (addr, admissibleAdditions sudoku addr)
   in  map adds
          (map fst (filter (isNothing . snd)
                           (Array.assocs sudoku)))



solutionTree :: T -> ATree T
solutionTree sudoku =
   let new (addr,elms) =
          map (\elm -> solutionTree (sudoku // [(addr, Just elm)])) elms
   in  ANode sudoku (map new (allAdmissibleAdditions sudoku))

treeAltToStandard :: ATree T -> Tree T
treeAltToStandard (ANode sudoku subs) =
   Tree.Node sudoku (concatMap (map treeAltToStandard) subs)

{- Convert a tree with alternatives for each position (ATree)
   into a normal tree by choosing one position and its alternative values.
   We need to consider only one position per level
   because the remaining positions are processed in the sub-levels.
   With other words: Choosing more than one position
   would lead to multiple reports of the same solution.

   For reasons of efficiency
   we choose the position with the least number of alternatives.
   If this number is zero, the numbers tried so far are wrong.
   If this number is one, then the choice is unique, but maybe still wrong.
   If the number of alternatives is larger,
   we have to check each alternative.
-}
treeAltToStandardOptimize :: ATree T -> Tree T
treeAltToStandardOptimize (ANode sudoku subs) =
   let chooseMinLen [] = []
       chooseMinLen xs = minimumBy compareLength xs
   in  Tree.Node sudoku (chooseMinLen
          (map (map treeAltToStandardOptimize) subs))

maybeComplete :: T -> Maybe Complete
maybeComplete sudoku =
   fmap (Array.array fieldBounds)
        (mapM (uncurry (fmap . (,))) (Array.assocs sudoku))

{- All leafs are at the same depth,
   namely the number of undetermined fields.
   That's why we can safely select all Sudokus at the lowest level. -}
solutions :: T -> [Complete]
solutions sudoku =
   let err = error "The lowest level should contain complete Sudokus only."
       {- "last'" is more efficient than "last" here
          because the program does not have to check
          whether deeper levels exist.
          We know that the tree is as deep
```

```
         as the number of undefined fields.
         This means that dropMatch returns a singleton list.
         We don't check that
         because then we would lose the efficiency again. -}
       last' = head . dropMatch (filter isNothing (Array.elems sudoku))
   in  map (fromMaybe err . maybeComplete)
           (last' (Tree.levels
             (treeAltToStandardOptimize (solutionTree sudoku))))



{- * transformations (can be used for construction, too) -}

standard :: Complete
standard =
   Array.listArray fieldBounds
      (map (\(i,j,k,l) -> mod (j+k) 3 * 3 + mod (i+l) 3 + 1)
           (range fieldBounds))


exampleHawiki0, exampleHawiki1 :: T
exampleHawiki0 = fromString (unlines [
      " 5  6   1",
      "  48   7 ",
      "8      52",
      "2   57 3 ",
      "         ",
      " 3 69   5",
      "79      8",
      " 1   65  ",
      "5   3  6 "
   ])

exampleHawiki1 = fromString (unlines [
      "    6  8 ",
      " 2       ",
      "  1      ",
      " 7    1 2",
      "5   3    ",
      "      4  ",
      "  42 1   ",
      "3  7  6  ",
      "       5 "
   ])



check :: Complete -> Bool
check sudoku =
   let checkParts select =
          all (\addr -> sort (select addr sudoku) == alphabet) squareRange
   in  all checkParts [selectRow, selectColumn, selectSquare]

selectRow, selectColumn, selectSquare ::
   (Coord,Coord) -> Array Address element -> [element]
selectRow (i,k) sudoku =
   map (sudoku!) (range ((i,0,k,0), (i,2,k,2)))
--   map (sudoku!) (map (\(j,l) -> (i,j,k,l)) squareRange)
selectColumn (j,l) sudoku =
   map (sudoku!) (range ((0,j,0,l), (2,j,2,l)))
```

```
selectSquare (i,j) sudoku =
   map (sudoku!) (range ((i,j,0,0), (i,j,2,2)))


{- * conversion from and to strings -}

put, putLn :: T -> IO ()
put   sudoku = putStr   (toString sudoku)
putLn sudoku = putStrLn (toString sudoku)

putC, putCLn :: Complete -> IO ()
putC   sudoku = putStr   (toString (fmap Just sudoku))
putCLn sudoku = putStrLn (toString (fmap Just sudoku))

fromString :: String -> T
fromString str =
   Array.array fieldBounds (concat
      (zipWith (\(i,k) -> map (\((j,l),x) -> ((i,j,k,l),x)))
         squareRange
         (map (zip squareRange . map charToElem) (lines str))))

toString :: T -> String
toString sudoku =
   unlines
      (map (\(i,k) -> map (\(j,l) -> elemToChar (sudoku!(i,j,k,l)))
                          squareRange)
           squareRange)

charToElem :: Char -> Maybe Element
charToElem c =
   toMaybe (isDigit c) (digitToInt c)

elemToChar :: Maybe Element -> Char
elemToChar =
   maybe ' ' intToDigit


{- * helper functions -}

nest :: Int -> (a -> a) -> a -> a
nest 0 _ x = x
nest n f x = f (nest (n-1) f x)

toMaybe :: Bool -> a -> Maybe a
toMaybe False _ = Nothing
toMaybe True  x = Just x

compareLength :: [a] -> [b] -> Ordering
compareLength (_:xs) (_:ys) = compareLength xs ys
compareLength []     []     = EQ
compareLength (_:_)  []     = GT
compareLength []     (_:_)  = LT

{- | Drop as many elements as the first list is long -}
dropMatch :: [b] -> [a] -> [a]
dropMatch xs ys =
   map fromJust (dropWhile isNothing
      (zipWith (toMaybe . null) (iterate (drop 1) xs) ys))
```

4 Exact set cover solver

By Henning Thielemann.

As part of the package set-cover (http://hackage.haskell.org/package/set-cover) there are two short solutions (http://code.haskell.org/~thielema/set-cover/example/Sudoku.hs) using a generic exact set cover solver.

One solution uses the `Set` type from the containers (http://hackage.haskell.org/package/containers) package whereas the other solution uses bit manipulation. Both solvers are pretty fast.

5 No guessing

By Simon Peyton Jones.

Since this page is here I thought I'd add a solver I wrote sometime last year. The main constraint I imposed is that it never guesses, and that it outputs a human-comprehensible explanation of every step of its reasoning. That means there are some puzzles it can't solve. I'd be interested to know if there are any puzzles that it gets stuck on where there is a no-guessing way forward. I made no attempt to make it fast.

There are two files: Media:SudokuPJ.hs and Media:TestPJ.hs. The latter just contains a bunch of test cases; I was too lazy to write a proper parser.

The main entry point is:

```
run1 :: Verbosity -> [String] -> Doc
data Verbosity = All | Terse | Final
```

The `[String]` the starting board configuration (see the tests file).

6 Just guessing

By ChrisKuklewicz

This solver is an implementation of Knuth's "Dancing Links" algorithm for solving binary-cover problems. This algorithm represents the constraints as a sparse binary matrix, with 1's as linked nodes. The nodes are in a vertical and a horizontal doubly linked list, and each vertical list is headed by another node that represents one of the constraints. It is interesting as an example of the rare beast in Haskell: a mutable data structure. The code has been rewritten and cleaned up here Media:DancingSudoku.lhs. Its main routine is designed to handle the input

from sudoku17 (http://www.csse.uwa.edu.au/~gordon/sudoku17) on stdin. Currently it only returns the first solution or calls an error, it can be modified (see comments in the file) to return all solutions in a list. An earlier version used ST.Lazy instead of ST.Strict which made operating on puzzles with many solutions more tractable.

Other trivia: It uses "mdo" and lazyness to initialize some of the doubly linked lists.

7 Very smart, with only a little guessing

by ChrisKuklewicz

This solver does its best to avoid the branch and guess approach. On the 36628 puzzles of length 17 (http://www.csse.uwa.edu.au/~gordon/sudokumin.php) it resorts to guessing on only 164. This extra strength comes from examining the constraints that can only be solved in exactly two ways, and how these constraints overlap and interact with each other and remaining possibilities.

The source code (http://evenmere.org/~chrisk/chris-sudoku-deduce.tar.gz) compiles to take a list of puzzles as input and produces a description of the number of (good and total) guesses required, as well as a shuffled version of the input. If there was guessing, then the shuffled version could be sent back into the solver to see how the difficulty depended on luck. The list of 164 hard puzzles is included with the source code. The Deduce.hs file contains comments.

The data is stored in a 9x9x9 boolean array, and the only operations are turning off possibilities and branching. For performance the array is thawed, mutated, and frozen. On the set of 36628 puzzles the speed averages 9.4 puzzles solved per second on a 1.33 GHz G4 (ghc-6.4.1 on OS X). I liked the 9x9x9 array since it emphasized the symmetry of the problem.

8 Only guessing without dancing links

by AndrewBromage

This solver (http://andrew.bromage.org/darcs/sudoku/) uses a different implementation of Knuth's algorithm, without using pointers. It instead relies on the fact that in Haskell, tree-like data structure (in this case, a Priority Search Queue) "undo" operations are essentially free.

9 Generalized solver

By Thorkil Naur

This Su Doku solver is able to solve classes of Su Doku puzzles that extend the ordinary 9*9 puzzles. The documentation describes the solver and also some (to the present author at least) surprising properties of various reduction strategies used when solving Su Doku puzzles.

The following files comprise the Su Doku solver and related code:

```
Media:Format.hs
Media:Merge.hs
Media:SdkMSol2.hs
Media:SortByF.hs
Media:SuDoku.hs
Media:t40.hs
Media:t44.hs
Media:Test.hs
```

For an example of use, the command

```
runhugs SdkMSol2 \
  tn1 \
  Traditional 3 \
  -#123456789 \
  1-53---9- \
  ---6----- \
  ------271 \
  82------- \
  ---487--- \
  ------53- \
  23------- \
  --7-59--- \
  --6---8-4
```

produces output that, among other things, contain

```
tn1: Solutions:
  1 7 5 3 2 8 4 9 6
  9 4 2 6 7 1 3 8 5
  3 6 8 5 9 4 2 7 1
  8 2 9 1 3 5 6 4 7
  6 5 3 4 8 7 9 1 2
  7 1 4 9 6 2 5 3 8
  2 3 1 8 4 6 7 5 9
  4 8 7 2 5 9 1 6 3
  5 9 6 7 1 3 8 2 4
```

10 Simple small solver

I haven't looked at the other solvers in detail yet, so I'm not sure what is good or bad about mine, but here it is:

```
http://darcs.brianweb.net/sudoku/Sudoku.pdf
```

```
http://darcs.brianweb.net/sudoku/src/Sudoku.lhs
```

-Brian Alliet <brian@brianweb.net>

11 Backtrack monad solver

This is a simple but fast solver that uses standard monads from the MonadTemplateLibrary in the StandardLibraries.

Besides being Yet Another Example of a Sudoko solver, I think it is also a nice somewhat-nontrivial example of monads in practice.

The idea is that the monad StateT s [] does backtracking. It means "iterate over a list while keeping state, but re-initialize to the original state on each iteration".

I have several (Unix command line) front-ends to this module, available upon request. The one I use most creates and prints six new Sudoku puzzles on a page, with fine-grain control over the difficulty of the puzzle. This has made me quite popular among friends and extended family.

```
- YitzGale
```

```haskell
{-# OPTIONS_GHC -fglasgow-exts #-}

-- Solve a Sudoku puzzle

module Sudoku where

import Control.Monad.State
import Data.Maybe (maybeToList)
import Data.List (delete)

type Value = Int
type Cell = (Int, Int) -- One-based coordinates

type Puzzle  = [[Maybe Value]]
type Solution = [[Value]]

-- The size of the puzzle.
sqrtSize :: Int
sqrtSize = 3
size = sqrtSize * sqrtSize

-- Besides the rows and columns, a Sudoku puzzle contains s blocks
-- of s cells each, where s = size.
blocks :: [[Cell]]
blocks = [[(x + i, y + j) | i <- [1..sqrtSize], j <- [1..sqrtSize]] |
          x <- [0,sqrtSize..size-sqrtSize],
          y <- [0,sqrtSize..size-sqrtSize]]

-- The one-based number of the block that a cell is contained in.
blockNum :: Cell -> Int
blockNum (row, col) = row - (row - 1) `mod` sqrtSize + (col - 1) `div` sqrtSize
```

```haskell
-- When a Sudoku puzzle has been partially filled in, the following
-- data structure represents the remaining options for how to proceed.
data Options = Options {
    cellOpts :: [[[Value]]], -- For each cell, a list of possible values
    rowOpts  :: [[[Cell ]]], -- For each row    and value, a list of cells
    colOpts  :: [[[Cell ]]], -- For each column and value, a list of cells
    blkOpts  :: [[[Cell ]]]  -- For each block  and value, a list of cells
  } deriving Show
modifyCellOpts f = do {opts <- get; put $ opts {cellOpts = f $ cellOpts opts}}
modifyRowOpts  f = do {opts <- get; put $ opts {rowOpts  = f $ rowOpts  opts}}
modifyColOpts  f = do {opts <- get; put $ opts {colOpts  = f $ colOpts  opts}}
modifyBlkOpts  f = do {opts <- get; put $ opts {blkOpts  = f $ blkOpts  opts}}

-- The full set of initial options, before any cells are constrained
initOptions :: Options
initOptions = Options {
  cellOpts = [[[1..size] | _ <- [1..size]] | _ <- [1..size]],
  rowOpts  = [[[(r, c)   | c <- [1..size]] | _ <- [1..size]] | r <- [1..size]],
  colOpts  = [[[(r, c)   | r <- [1..size]] | _ <- [1..size]] | c <- [1..size]],
  blkOpts  = [[b         | _ <- [1..size]] | b <- blocks]}

solve :: Puzzle -> [Solution]
solve puz = evalStateT (initPuzzle >> solutions) initOptions
  where
    initPuzzle =
      sequence_ [fixCell v (r, c) | (row, r) <- zip puz [1..],
                                    (val, c) <- zip row [1..],
                                    v <- maybeToList val]

-- Build a list of all possible solutions given the current options.
-- We use a list monad INSIDE a state monad. That way,
-- the state is re-initialized on each element of the list iteration,
-- allowing backtracking when an attempt fails (with mzero).
solutions :: StateT Options [] Solution
solutions = solveFromRow 1
  where
    solveFromRow r
     | r > size  = return []
     | otherwise = do
         row  <- solveRowFromCol r 1
         rows <- solveFromRow $ r + 1
         return $ row : rows
    solveRowFromCol r c
     | c > size  = return []
     | otherwise = do
         vals <- gets $ (!! (c - 1)) . (!! (r - 1)) . cellOpts
         val <- lift vals
         fixCell val (r, c)
         row <- solveRowFromCol r (c + 1)
         return $ val : row

-- Fix the value of a cell.
-- More specifically - update Options to reflect the given value at
-- the given cell, or mzero if that is not possible.
fixCell :: (MonadState Options m, MonadPlus m) =>
           Value -> Cell -> m ()
fixCell val cell@(row, col) = do
    vals <- gets $ (!! (col - 1)) . (!! (row - 1)) . cellOpts
    guard $ val `elem` vals
    modifyCellOpts $ replace2 row col [val]
```

```
    modifyRowOpts  $ replace2 row val [cell]
    modifyColOpts  $ replace2 col val [cell]
    modifyBlkOpts  $ replace2 blk val [cell]
    sequence_ [constrainCell v   cell     | v <- [1..size], v /= val]
    sequence_ [constrainCell val (row, c) | c <- [1..size], c /= col]
    sequence_ [constrainCell val (r, col) | r <- [1..size], r /= row]
    sequence_ [constrainCell val c | c <- blocks !! (blk - 1), c /= cell]
  where
    blk = blockNum cell

-- Assert that the given value cannot occur in the given cell.
-- Fail with mzero if that means that there are no options left.
constrainCell :: (MonadState Options m, MonadPlus m) =>
                 Value -> Cell -> m ()
constrainCell val cell@(row, col) = do
    constrainOpts row col val  cellOpts modifyCellOpts (flip fixCell cell)
    constrainOpts row val cell rowOpts  modifyRowOpts  (fixCell val)
    constrainOpts col val cell colOpts  modifyColOpts  (fixCell val)
    constrainOpts blk val cell blkOpts  modifyBlkOpts  (fixCell val)
  where
    blk = blockNum cell
    constrainOpts x y z getOpts modifyOpts fixOpts = do
      zs <- gets $ (!! (y - 1)) . (!! (x - 1)) . getOpts
      case zs of
        [z']  -> guard (z' /= z)
        [_,_] -> when (z `elem` zs) $ fixOpts (head $ delete z zs)
        (_:_) -> modifyOpts $ replace2 x y (delete z zs)
        _     -> mzero

-- Replace one element of a list.
-- Coordinates are 1-based.
replace :: Int -> a -> [a] -> [a]
replace i x (y:ys)
 | i > 1     = y : replace (i - 1) x ys
 | otherwise = x : ys
replace _ _ _ = []

-- Replace one element of a 2-dimensional list.
-- Coordinates are 1-based.
replace2 :: Int -> Int -> a -> [[a]] -> [[a]]
replace2 i j x (y:ys)
 | i > 1     = y : replace2 (i - 1) j x ys
 | otherwise = replace j x y : ys
replace2 _ _ _ _ = []
```

12 In-flight entertainment

By Lennart Augustsson

When on a Lufthansa trans-atlantic flight in 2005 I picked up the in-flight magazine and found a Sudoku puzzle. I decided to finally try one. After solving half of it by hand I got bored. Surely, this mechanical task is better performed by a machine? So I pulled out my laptop and wrote a Haskell program.

The program below is what I wrote on the plane, except for some comments that

I've added. I have made no attempt as making it fast, so the nefarious test puzzle below takes a minute to solve.

First, the solver without user interface:

```
module Sudoku(Square, Board, ColDigit, RowDigit, BoxDigit, Digit, initialBoard, getBoard, mkS
import Char(intToDigit, digitToInt)
import List ((\\), sortBy)

-- A board is just a list of Squares.  It always has all the squares.
data Board = Board [Square]
        deriving (Show)

-- A Square contains its column (ColDigit), row (RowDigit), and
-- which 3x3 box it belongs to (BoxDigit).  The box can be computed
-- from the row and column, but is kept for speed.
-- A Square also contains it's status: either a list of possible
-- digits that can be placed in the square OR a fixed digit (i.e.,
-- the square was given by a clue or has been solved).
data Square = Square ColDigit RowDigit BoxDigit (Either [Digit] Digit)
        deriving (Show)

type ColDigit = Digit
type RowDigit = Digit
type BoxDigit = Digit
type Digit = Char       -- '1' .. '9'

-- The initial board, no clues given so all digits are possible in all squares.
initialBoard :: Board
initialBoard = Board [ Square col row (boxDigit col row) (Left allDigits) |
                       row <- allDigits, col <- allDigits ]

-- Return a list of rows of a solved board.
-- If used on an unsolved board the return value is unspecified.
getBoard :: Board -> [[Char]]
getBoard (Board sqs) = [ [ getDigit d | Square _ row' _ d <- sqs, row' == row ] | row <- allD
  where getDigit (Right d) = d
        getDigit _ = '0'

allDigits :: [Char]
allDigits = ['1' .. '9']

-- Compute the box from a column and row.
boxDigit :: ColDigit -> RowDigit -> BoxDigit
boxDigit c r = intToDigit $ (digitToInt c - 1) `div` 3 + (digitToInt r - 1) `div` 3 * 3 + 1

-- Given a column, row, and a digit make a (solved) square representing this.
mkSquare :: ColDigit -> RowDigit -> Digit -> Square
mkSquare col row c | col `elem` allDigits && row `elem` allDigits && c `elem` allDigits
                   = Square col row (boxDigit col row) (Right c)
mkSquare _ _ _ = error "Bad mkSquare"

-- Place a given Square on a Board and return the new Board.
-- Illegal setSquare calls will just error out.  The main work here
-- is to remove the placed digit from the other Squares on the board
-- that are in the same column, row, or box.
setSquare :: Square -> Board -> Board
setSquare sq@(Square scol srow sbox (Right d)) (Board sqs) = Board (map set sqs)
  where set osq@(Square col row box ds) =
            if col == scol && row == srow then sq
```

```
            else if col == scol || row == srow || box == sbox then (Square col row box (sub d
            else osq
        sub (Left ds) = Left (ds \\ [d])
        sub (Right d') | d == d' = error "Impossible setSquare"
        sub dd = dd
setSquare _ _ = error "Bad setSquare"

-- Get the unsolved Squares from a Board.
getLeftSquares :: Board -> [Square]
getLeftSquares (Board sqs) = [ sq | sq@(Square _ _ _ (Left _)) <- sqs ]

-- Given an initial Board return all the possible solutions starting
-- from that Board.
-- Note, this all happens in the list monad and makes use of lazy evaluation
-- to avoid work.  Using the list monad automatically handles all the backtracking
-- and enumeration of solutions.
solveMany :: Board -> [Board]
solveMany brd =
    case getLeftSquares brd of
    [] -> return brd            -- Nothing unsolved remains, we are done.
    sqs -> do
        -- Sort the unsolved Squares by the ascending length of the possible
        -- digits.  Pick the first of those so we always solve forced Squares
        -- first.
        let Square c r b (Left ds) : _ = sortBy leftLen sqs
            leftLen (Square _ _ _ (Left ds1)) (Square _ _ _ (Left ds2)) = compare (length ds1
            leftLen _ _ = error "bad leftLen"
        sq <- [ Square c r b (Right d) | d <- ds ] -- Try all possible moves
        solveMany (setSquare sq brd) -- And solve the extended Board.
```

Second, a simple user interface (a different user interface that I have is an Excell addin):

```
module Main where
import Sudoku

--         Col    Row   Digit
solve :: [((Char, Char), Char)] -> [[Char]]
solve crds =
    let brd = foldr add initialBoard crds
        add ((c, r), d) = setSquare (mkSquare c r d)
    in  case solveMany brd of
        [] -> error "No solutions"
        b : _ -> getBoard b

-- The parse assumes that squares without a clue
-- contain '0'.
main = interact $
     unlines .                                          -- turn it into lines
     map (concatMap (:" ")) .                           -- add a space after each digit fo
     solve .                                            -- solve the puzzle
     filter ((`elem` ['1'..'9']) . snd) .               -- get rid of non-clues
     zip [ (c, r) | r <- ['1'..'9'], c <- ['1'..'9'] ] .   -- pair up the digits with their c
     filter (`elem` ['0'..'9'])                         -- get rid of non-digits
```

13 Sudoku incrementally, à la Bird

> As part of a new Advanced Functional Programming (http://cs.nott.ac.uk/~gmh/afp.html) course in Nottingham, Graham Hutton (http://cs.nott.ac.uk/~gmh/) presented a Haskell approach to solving Sudoku puzzles, based upon notes from Richard Bird (http://web.comlab.ox.ac.uk/oucl/work/richard.bird/) . The approach is classic Bird: start with a simple but impractical solver, whose efficiency is then improved in a series of steps. The end result is an elegant program that is able to solve any Sudoku puzzle in an instant. Its also an excellent example of what has been termed wholemeal programming focusing on entire data structures rather than their elements. (Transplanted from LtU (http://lambda-the-ultimate.org/node/772) .)

A full talk-through of the evolution of the code may be found under the course page (http://cs.nott.ac.uk/~gmh/sudoku.lhs) . --Liyang 13:35, 27 July 2006 (UTC)

I've also written Media:sudokuWss.hs, a parallel version of this solver. It uses STM to prune the boxes, columns, and rows simultaneously, which is kind of cool. I'm pretty sure it can be optimized quite a bit... --WouterSwierstra, August 2007.

14 607 bytes / 12 lines

A super quick attempt at a smallest solution, based on the 707 byte sudoku (http://web.math.unifi.it/users/maggesi/haskell_sudoku_solver.html) solver:

```
import List

main = putStr . unlines . map disp . solve . return . input =<< getContents

solve s = foldr (\p l -> [mark (p,n) s | s <- l, n <- s p]) s idx

mark (p@(i,j),n) s q@(x,y)
    | p == q                              = [n]
    | x == i || y == j || e x i && e y j = delete n (s q)
    | otherwise                           = s q
    where e a b = div (a-1) 3 == div (b-1) 3

disp s = unlines [unwords [show $ head $ s (i,j) | j <- [1..9]] | i <- [1..9]]

input s = foldr mark (const [1..9]) $
  [(p,n) | (p,n) <- zip idx $ map read $ lines s >>= words, n>0]

idx = [(i,j) | i <- [1..9], j <- [1..9]]
```

dons 07:54, 2 December 2006 (UTC)

15 A parallel solver

A parallel version of Richard Bird's function pearl solver by Wouter Swierstra:

http://www.haskell.org/sitewiki/images/1/12/SudokuWss.hs

16 Another simple solver

One day I wrote a completely naive sudoku solver which tried all possibilities to try arrays in Haskell. It works, however I doubt that I'll see it actually solve a puzzle during my remaining lifetime.

So I set out to improve it. The new version still tries all possibilities, but it starts with the cell that has a minimal number of possibilities.

```
import Array
import List
import System

-- ([Possible Entries], #Possible Entries)
type Field = Array (Int,Int) ([Int], Int)

-- Fields are Strings of Numbers with 0 in empty cells
readField ::String -> Field
readField f =  listArray ((1,1),(9,9)) (map (\j -> let n=read [j]::Int in if n==0 then ([0..9

-- x y wrong way -> reading wrong? no effect on solution though
showField :: Field -> String
showField f = unlines [concat [show $ entry (f!(y,x))|x<-[1..9]]|y<-[1..9]]

printField :: Maybe Field -> String
printField (Just f) = concat [concat [show $ entry f!(y,x))|x<-[1..9]]|y<-[1..9]]
printField Nothing = "No solution"

-- true if cell is empty
isEmpty :: ([Int],Int) -> Bool
isEmpty (xs,_) = xs == [0]

entry :: ([Int],Int) -> Int
entry = head.fst

-- 0 possibilties left, no emtpy fields
done :: Field -> Bool
done a = let l=elems a in 0==foldr (\(_,x) y -> x+y) 0 l && all (not.isEmpty) l

--return column/row/square containing coords (x,y), excluding (x,y)
column::Field ->(Int,Int) -> [Int]
column a ~(x,y)= [entry $ a!(i,y)|i<-[1..9],i/=x]

row :: Field -> (Int,Int) -> [Int]
row a ~(x,y)= [entry $ a!(x,j)|j<-[1..9],j/=y]

square :: Field -> (Int, Int)-> [Int]
square a ~(x,y) = block
        where
         n = head $ dropWhile (<x-3) [0,3,6]
         m = head $ dropWhile (<y-3) [0,3,6]
         block = [entry $ a!(i+n,j+m)|i<-[1..3],j<-[1..3],x/=i+n || y/=j+m]

-- remove invalid possibilities
remPoss :: Field -> Field
remPoss f =array ((1,1),(9,9)) $ map remPoss' (assocs f)
        where
```

```
        others xy= filter (/=0) $ row f xy ++ column f xy ++ square f xy
        remPoss' ~(i,(xs,n))
               | n/=0 = let nxs= filter ( `notElem` others i ) xs in (i,(nxs,length $ filter
               | otherwise = (i,(xs,n))

-- remove invalid fields, i.e. contains empty cell without filling possibilities
remInv :: [Field] -> [Field]
remInv = filter (all (\(_,(x,_)) -> x/=[0]).assocs)


genMoves :: (Int,Int) -> Field -> [Field]
genMoves xy f = remInv $ map remPoss [f // [(xy,([poss!!i],0))]|i<-[0..num-1]]
        where
         poss = tail $ fst (f!xy)
         num = snd (f!xy)

--always try the entry with least possibilties first
moves :: Field -> [Field]
moves f = genMoves bestOne f
        where
         -- remove all with 0 possibilities, select the one with minimum possibilities
         bestOne =fst $ minimumBy (\(_,(_,n)) (_,(_,m)) -> compare n m) list
         list = ((filter (\(_,(_,x)) -> x/=0).assocs) f)

play :: [Field] -> Maybe Field
play (f:a)
        | done f= Just f
        | otherwise = play (moves f++a)
play [] = Nothing

-- reads a file with puzzles, path as argument
main :: IO ()
main = do
                path <- getArgs
                inp<-readFile (path!!0)
                let x=lines inp
                let erg=map (printField.play) (map ((\x->[x]).remPoss.readField) x)
                writeFile "./out.txt" (unlines erg)
```

I let it run on the 41747 minimal puzzles. On a 2.66 GHz Intel Xeon it took 15441m1.920s, which is about 22 seconds per puzzle. It could probably be further improved by making remPoss smarter. At the time of writing this the naive version from which I started is crunching for 20 *days* on a simple puzzle with 32 hints. I'd say that's quite a performance improvement.

17 Constraint Propagation (a la Norvig)

By Manu

This is an Haskell implementation of Peter Norvig's sudoku solver (http://norvig.com/sudoku.html). It should solve, in a flash, the 95 puzzles found here : http://norvig.com/top95.txt Thanks to Daniel Fischer for helping and refactoring.

```haskell
module Main where

import Data.List hiding (lookup)
import Data.Array
import Control.Monad
import Data.Maybe

-- Types
type Digit  = Char
type Square = (Char,Char)
type Unit   = [Square]

-- We represent our grid as an array
type Grid = Array Square [Digit]


-- Setting Up the Problem
rows = "ABCDEFGHI"
cols = "123456789"
digits = "123456789"
box = (('A','1'),('I','9'))

cross :: String -> String -> [Square]
cross rows cols = [ (r,c) | r <- rows, c <- cols ]

squares :: [Square]
squares = cross rows cols  -- [('A','1'),('A','2'),('A','3'),...]

peers :: Array Square [Square]
peers = array box [(s, set (units!s)) | s <- squares ]
      where
        set = nub . concat

unitlist :: [Unit]
unitlist = [ cross rows [c] | c <- cols ] ++
           [ cross [r] cols | r <- rows ] ++
           [ cross rs cs | rs <- ["ABC","DEF","GHI"],
                           cs <- ["123","456","789"]]

-- this could still be done more efficiently, but what the heck...
units :: Array Square [Unit]
units = array box [(s, [filter (/= s) u | u <- unitlist, s `elem` u ]) |
                   s <- squares]


allPossibilities :: Grid
allPossibilities = array box [ (s,digits) | s <- squares ]

-- Parsing a grid into an Array
parsegrid     :: String -> Maybe Grid
parsegrid g    = do regularGrid g
                    foldM assign allPossibilities (zip squares g)

   where  regularGrid   :: String -> Maybe String
          regularGrid g  = if all (`elem` "0.-123456789") g
                           then Just g
                           else Nothing

-- Propagating Constraints
assign        :: Grid -> (Square, Digit) -> Maybe Grid
assign g (s,d) = if d `elem` digits
```

```haskell
                -- check that we are assigning a digit and not a '.'
                 then do
                   let ds = g ! s
                       toDump = delete d ds
                   foldM eliminate g (zip (repeat s) toDump)
                 else return g

eliminate     ::  Grid -> (Square, Digit) -> Maybe Grid
eliminate g (s,d) =
  let cell = g ! s in
  if d `notElem` cell then return g -- already eliminated
  -- else d is deleted from s' values
    else do let newCell = delete d cell
                newV = g // [(s,newCell)]
            newV2 <- case newCell of
            -- contradiction : Nothing terminates the computation
                 []   -> Nothing
            -- if there is only one value left in s, remove it from peers
                 [d'] -> do let peersOfS = peers ! s
                            foldM eliminate newV (zip peersOfS (repeat d'))
            -- else : return the new grid
                 _    -> return newV
            -- Now check the places where d appears in the peers of s
            foldM (locate d) newV2 (units ! s)

locate :: Digit -> Grid -> Unit -> Maybe Grid
locate d g u = case filter ((d `elem`) . (g !)) u of
               []  -> Nothing
               [s] -> assign g (s,d)
               _   -> return g

-- Search
search :: Grid -> Maybe Grid
search g =
  case [(l,(s,xs)) | (s,xs) <- assocs g, let l = length xs, l /= 1] of
            [] -> return g
            ls -> do let (_,(s,ds)) = minimum ls
                     msum [assign g (s,d) >>= search | d <- ds]

solve :: String -> Maybe Grid
solve str = do
    grd <- parsegrid str
    search grd

-- Display solved grid
printGrid :: Grid -> IO ()
printGrid = putStrLn . gridToString

gridToString :: Grid -> String
gridToString g =
  let l0 = elems g
      -- [("1537"),("4"),...]
      l1 = (map (\s -> " " ++ s ++ " ")) l0
      -- ["1 "," 2 ",...]
      l2 = (map concat . sublist 3) l1
      -- ["1  2  3 "," 4  5  6 ", ...]
      l3 = (sublist 3) l2
      -- [["1  2  3 "," 4  5  6 "," 7  8  9 "],...]
      l4 = (map (concat . intersperse "|")) l3
      -- ["1  2  3 | 4  5  6 | 7  8  9 ",...]
      l5 = (concat . intersperse [line] . sublist 3) l4
```

```
  in unlines l5
     where sublist n [] = []
           sublist n xs = ys : sublist n zs
             where (ys,zs) = splitAt n xs
           line = hyphens ++ "+" ++ hyphens ++ "+" ++ hyphens
           hyphens = replicate 9 '-'

main :: IO ()
main = do
    grids <- fmap lines $ readFile "top95.txt"
    mapM_ printGrid $ mapMaybe solve grids
```

18 Concurrent STM Solver

Liyang wrote some applicative functor porn utilising STM. It's pretty but slow. Suggestions for speeding it up would be very welcome.

19 Chaining style Solver

```
by jinjing
```

It uses some snippets (http://github.com/nfjinjing/projectt/tree/master/T/Snippets.hs) and the dot hack (http://github.com/nfjinjing/projectt/tree/master/T/Hack/Dot.hs)

```
import Prelude hiding ((.))
import T.T
import List
import Data.Maybe
import Data.Char
import Data.Map(keys, elems)
import qualified Data.Map as Map

row i = i `div` 9
col i = i `mod` 9
row_list i positions = positions.select(on_i_row) where
  on_i_row pos = pos.row  == i.row
col_list i positions = positions.select(on_i_col) where
  on_i_col pos = pos.col == i.col

grid_list i positions = positions.select(on_same_grid i)

on_same_grid i j = on_same_row_grid i j && on_same_col_grid i j

on_same_row_grid i j = ( i.row.mod.send_to(3) - j.row.mod.send_to(3) ) == i.row - j.row
on_same_col_grid i j = ( i.col.mod.send_to(3) - j.col.mod.send_to(3) ) == i.col - j.col

board = 0.upto 80
choices = 1.upto 9

related i positions =
  positions.row_list(i) ++ positions.col_list(i) ++ positions.grid_list(i)
values moves positions = positions.mapMaybe (moves.let_receive Map.lookup)
```

```
possible_moves i moves =
  let positions = moves.keys in
    choices \\ positions.related(i).values(moves)

sudoku_move moves =
  let i = moves.next_pos in
    moves.possible_moves(i).map(Map.insert i).map_send_to(moves)

next_pos moves = (board \\ moves.keys)
  .label_by(choice_size).sort.first.snd where
    choice_size i = moves.possible_moves(i).length

solve solutions 0 = solutions
solve solutions n = solve next_solutions (n-1) where
  next_solutions = solutions.map(sudoku_move).concat

parse_input line = line.words.join("")
  .map(\c -> if '1' <= c && c <= '9' then c else '0')
  .map(digitToInt).zip([0..]).reject((==0).snd).Map.fromList

pretty_output solution = solution.elems.map(show).in_group_of(9)
  .map(unwords).unlines

sudoku line = solve [given] (81 - given.Map.size).first.pretty_output
  where given = parse_input line
```

20 Finite Domain Constraint Solver

by David Overton

This solver uses a finite domain constraint solver monad described here (http://overtond.blogspot.com/2008/07/pre.html) . The core functions are shown below. A full explanation is here (http://overtond.blogspot.com/2008/07/haskell-sudoku-solver-using-finite.html) .

```
type Puzzle = [Int]

sudoku :: Puzzle -> [Puzzle]
sudoku puzzle = runFD $ do
    vars <- newVars 81 [1..9]
    zipWithM_ (\x n -> when (n > 0) (x `hasValue` n)) vars puzzle
    mapM_ allDifferent (rows vars)
    mapM_ allDifferent (columns vars)
    mapM_ allDifferent (boxes vars)
    labelling vars

rows, columns, boxes :: [a] -> [[a]]
rows = chunk 9
columns = transpose . rows
boxes = concatMap (map concat . transpose) . chunk 3 . chunk 3 . chunk 3

chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = ys : chunk n zs where
    (ys, zs) = splitAt n xs
```

21 Very fast Solver

by Frank Kuehnel

This solver implements constraint propagation with higher level logic and search. Solves the 49151 puzzles with 17 hints in less than 50 seconds! More detail and less optimized versions are here (http://zufaellige-reflektion.blogspot.com/2011/01/sudoku-puzzels-revisited.html) .

```
module Main where

import qualified Data.Vector.Unboxed as V
import qualified Data.Vector as BV (generate,(!))
import Data.List (foldl',sort,group)
import Data.Char (chr, ord)
import Data.Word
import Data.Bits
import Control.Monad
import Data.Maybe
import System (getArgs)

-- Types
type Alphabet   = Word8
type Hypothesis = Word32

-- Hypotheses space is a matrix of independed hypoteses
type HypothesesSpace    = V.Vector Hypothesis

-- Set up spatial transformers / discriminators to reflect the spatial
-- properties of a Sudoku puzzle
ncells = 81

-- vector rearrangement functions
rows        :: HypothesesSpace -> HypothesesSpace
rows        = id

columns     :: HypothesesSpace -> HypothesesSpace
columns vec = V.map (\cidx -> vec `V.unsafeIndex` cidx) cIndices
            where cIndices = V.fromList [r*9 + c | c <-[0..8], r<-[0..8]]

subGrids    :: HypothesesSpace -> HypothesesSpace
subGrids vec= V.map (\idx -> vec `V.unsafeIndex` idx) sgIndices
            where sgIndices = V.fromList [i + bc + br | br <- [0,27,54], bc <- [0,3,6], i<-[0

-- needs to be boxed, because vector elements are not primitives
peersDiscriminators = BV.generate ncells discriminator
        where
            discriminator idx1 = V.zipWith3 (\r c s -> (r || c || s)) rDscr cDscr sDscr
                where
                    rDscr = V.generate ncells (\idx2 -> idx1 `div` 9 == idx2 `div` 9)
                    cDscr = V.generate ncells (\idx2 -> idx1 `mod` 9 == idx2 `mod` 9)
                    sDscr = V.generate ncells (\idx2 -> subGridOfidx1 == subGrid idx2)
                        where
                            subGridOfidx1 = subGrid idx1
                            subGrid idx = (idx `div` 27, (idx `div` 3) `mod` 3)

-- Let's implement the logic
```

```haskell
-- Level 0 logic (enforce consistency):
-- We can't have multiple same solutions in a peer unit,
-- eliminate solutions from other hypotheses
enforceConsistency :: HypothesesSpace -> Maybe HypothesesSpace
enforceConsistency hypS0 = do
        V.foldM solutionReduce hypS0 $ V.findIndices newSingle hypS0

solutionReduce :: HypothesesSpace -> Int -> Maybe HypothesesSpace
solutionReduce hypS0 idx =
        let sol     = hypS0 V.! idx
            peers   = peersDiscriminators BV.! idx
            hypS1   = V.zipWith reduceInUnit peers hypS0
                where
                    reduceInUnit p h
                                | p && (h == sol)   = setSolution sol
                                | p                 = h `minus` sol
                                | otherwise         = h
        in if V.any empty hypS1
            then return hypS1
            else if V.any newSingle hypS1
                    then enforceConsistency hypS1 -- constraint propagation
                    else return hypS1

-- Level 1 logic is rather simple:
-- We tally up all unknown values in a given unit,
-- if a value occurs only once, then it must be the solution!
localizeSingles :: HypothesesSpace -> Maybe HypothesesSpace
localizeSingles unit = let known = maskChoices $ accumTally $ V.filter single unit
        in if dups known
            then Nothing
            else
                case (filterSingles $ accumTally $ V.filter (not . single) unit) `minus` know
                    0       -> return unit
                    sl      -> return $ replaceWith unit sl
                        where
                            replaceWith :: V.Vector Hypothesis -> Hypothesis -> V.Vector Hypo
                            replaceWith unit s = V.map (\u -> if 0 /= maskChoices (s .&. u) t

-- Level 2 logic is a bit more complicated:
-- Say in a given unit, we find exactly two places with the hypothesis {1,9}.
-- Then obviously, the value 1 and 9 can only occur in those two places.
-- All other ocurrances of the value 1 and 9 can eliminated.
localizePairs :: HypothesesSpace -> Maybe HypothesesSpace
localizePairs unit = let pairs = V.toList $ V.filter pair unit
        in if nodups pairs
            then return unit
            else
                case map head $ filter lpair $ tally pairs of
                    []          ->  return unit
                    pl@(p:ps)   ->  return $ foldl' eliminateFrom unit pl
                        where -- "subtract" pair out of a hypothesis
                            eliminateFrom :: V.Vector Hypothesis -> Hypothesis -> V.Vector Hy
                            eliminateFrom unit p = V.map (\u -> if u /= p then u `minus` p el

-- Level 3 logic resembles the level 2 logic:
-- If we find exactly three places with the hypothesis {1,7,8} in a given unit, then all othe
-- you'll get the gist!
localizeTriples :: HypothesesSpace -> Maybe HypothesesSpace
localizeTriples unit = let triples = V.toList $ V.filter triple unit
        in if nodups triples
```

```
            then return unit
            else
                case map head $ filter ltriple $ tally triples of
                    []          ->  return unit
                    tl@(t:ts)   ->  return $ foldl' eliminateFrom unit tl
                        where -- "subtract" triple out of a hypothesis
                            eliminateFrom :: V.Vector Hypothesis -> Hypothesis -> V.Vector Hy
                            eliminateFrom unit t = V.map (\u -> if u /= t then u `minus` t el

-- Even higher order logic is easy to implement, but becomes rather useless in the general ca

-- Implement the whole nine yard: constraint propagation and search

applySameDimensionLogic :: HypothesesSpace -> Maybe HypothesesSpace
applySameDimensionLogic hyp0 = do
        res1 <- logicInDimensionBy rows chainedLogic hyp0
        res2 <- logicInDimensionBy columns chainedLogic res1
        logicInDimensionBy subGrids chainedLogic res2
            where
                chainedLogic = localizeSingles >=> localizePairs >=> localizeTriples

logicInDimensionBy :: (HypothesesSpace -> HypothesesSpace) -> (HypothesesSpace -> Maybe Hypot
logicInDimensionBy trafo logic hyp = liftM (trafo . V.concat) $ mapM (\ridx -> do logic $ V.u
        where
            hyp' :: HypothesesSpace
            hyp' = trafo hyp

prune :: HypothesesSpace -> Maybe HypothesesSpace
prune hypS0 = do
        hypS1 <- applySameDimensionLogic =<< enforceConsistency hypS0
        if V.any newSingle hypS1
            then prune hypS1    -- effectively implemented constraint propagation
            else do
                hypS2 <- applySameDimensionLogic hypS1
                if hypS1 /= hypS2
                    then prune hypS2    -- effectively implemented a fix point method
                    else return hypS2

search :: HypothesesSpace -> Maybe HypothesesSpace
search hypS0
        | complete hypS0    = return hypS0
        | otherwise         = do msum [prune hypS1 >>= search | hypS1 <- expandFirst hypS0]

-- guessing order makes a big difference!!
expandFirst :: HypothesesSpace -> [HypothesesSpace]
expandFirst hypS
        | suitable == []    = []
        | otherwise         = let (_, idx) = minimum suitable -- minimum is the preferred str
                              in map (\choice -> hypS V.// [(idx, choice)]) (split $ hypS V.!
    where
        suitable = filter ((>1) . fst) $ V.toList $ V.imap (\idx e -> (numChoices e, idx)) hy

-- Some very useful tools:
-- partition a list into sublists
chop            :: Int -> [a] -> [[a]]
chop n []       =  []
chop n xs       =  take n xs : chop n (drop n xs)

-- when does a list have no duplicates
nodups          :: Eq a => [a] -> Bool
nodups []       =  True
```

```haskell
nodups (x:xs)   =  not (elem x xs) && nodups xs

dups            :: Hypothesis -> Bool
dups t          = (filterDups t) /= 0

tally           :: Ord a => [a] -> [[a]]
tally           = group . sort

empty           :: Hypothesis -> Bool
empty n         = (maskChoices n) == 0

single          :: Hypothesis -> Bool
single n        = (numChoices n) == 1

lsingle         :: [a] -> Bool
lsingle [n]     = True
lsingle _       = False

pair            :: Hypothesis -> Bool
pair n          = numChoices n == 2

lpair           :: [a] -> Bool
lpair (x:xs)    = lsingle xs
lpair _         = False

triple          :: Hypothesis -> Bool
triple n        = (numChoices n) == 3

ltriple         :: [a] -> Bool
ltriple (x:xs)  = lpair xs
ltriple _       = False

complete        :: HypothesesSpace -> Bool
complete        = V.all single

-- The bit gymnastics (wish some were implemented in Data.Bits)
-- bits 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 .. 27 28 29 30 31 represents
--      h - h - h - h - h - h  -  h  -  h  -  h  -  .. s  l  l  l  l
-- with
--      h : 1 iff element is part of the hypothesis set
--      l : 4 bits for the cached number of h bits set
--      s : 1 iff a single solution for the cell is found

-- experiment with different strategies
split           :: Hypothesis -> [Hypothesis]
split 0         = []
split n         = [n `minus` bit1, (bit 28) .|. bit1]
    where bit1 = (bit $ firstBit n)

minus           :: Hypothesis -> Hypothesis -> Hypothesis
xs `minus` ys
        | maskChoices (xs .&. ys) == 0  = xs
        | otherwise = zs .|. ((countBits zs) `shiftL` 28)
                        where zs = maskChoices $ xs .&. (complement ys)

numChoices      :: Hypothesis -> Word32
numChoices n    = (n `shiftR` 28)

newSingle       :: Hypothesis -> Bool
newSingle n     = (n `shiftR` 27) == 2
```

```haskell
isSolution      :: Hypothesis -> Bool
isSolution n    = n `testBit` 27

setSolution     :: Hypothesis -> Hypothesis
setSolution n   = n `setBit` 27

maskChoices     :: Hypothesis -> Hypothesis
maskChoices n   =  n .&. 0x07FFFFFF

intersect       :: Hypothesis -> Hypothesis -> Hypothesis
intersect x y   = z .|. ((countBits z) `shiftL` 28)
                    where z = maskChoices $ x .&. y

countBits       :: Word32 -> Word32 -- would be wonderful if Data.Bits had such a function
countBits 0     = 0
countBits n     = (cBLH 16 0xFFFF . cBLH 8 0xFF00FF . cBLH 4 0x0F0F0F0F . cBLH 2 0x33333333 .
cBLH            :: Int -> Word32 -> Word32 -> Word32
cBLH s mask n   = (n .&. mask) + (n `shiftR` s) .&. mask

firstBit        :: Hypothesis -> Int -- should also be in Data.Bits
firstBit 0      = 0 -- stop recursion !!
firstBit n
        | n .&. 1 > 0       = 0
        | otherwise         = (+) 1 $ firstBit $ n `shiftR` 1

accumTally      :: V.Vector Hypothesis -> Hypothesis
accumTally nl   = V.foldl' accumTally2 0 nl
accumTally2     :: Word32 -> Word32 -> Word32
accumTally2 t n = (+) t $ n .&. (((complement t) .&. 0x02AAAAAA) `shiftR` 1)

filterSingles   :: Hypothesis -> Hypothesis
filterSingles t = t .&. (((complement t) .&. 0x02AAAAAA) `shiftR` 1)

filterDups      :: Hypothesis -> Hypothesis
filterDups t    = (t .&. 0x02AAAAAA) `shiftR` 1

defaultHypothesis :: Hypothesis
defaultHypothesis = 0x90015555 -- all nine alphabet elements are set

mapAlphabet :: V.Vector Hypothesis
mapAlphabet = V.replicate 256 defaultHypothesis V.// validDigits
    where
        validDigits :: [(Int, Hypothesis)]
        validDigits = [(ord i, (bit 28) .|. (bit $ 2*(ord i - 49))) | i <- "123456789"]

toChar :: Hypothesis -> [Char]
toChar s
        | single s      = [normalize s]
        | otherwise     = "."
        where
            normalize s = chr $ (+) 49 $ (firstBit s) `shiftR` 1

toCharDebug :: Hypothesis -> [Char]
toCharDebug s
        | isSolution s          = ['!', normalize s]
        | single s              = [normalize s]
        | otherwise             = "{" ++ digits ++ "}"
        where
            normalize s = chr $ (+) 49 $ (firstBit s) `shiftR` 1
            digits = zipWith test "123456789" $ iterate (\e -> e `shiftR` 2) s
            test c e
```

```
                    | e.&.1 == 1    = c
                    | otherwise     = '.'

-- Initial hypothesis space
initialize :: String -> Maybe HypothesesSpace
initialize g =  if all (`elem` "0.-123456789") g
                then
                    let
                        hints = zip [0..] translated
                        translated = map  (\c -> mapAlphabet V.! ord c) $ take ncells g
                    in Just $ (V.replicate ncells defaultHypothesis) V.// hints
                else Nothing

-- Display (partial) solution
printResultD :: HypothesesSpace -> IO ()
printResultD = putStrLn . toString
    where
        toString :: HypothesesSpace -> String
        toString hyp = unlines $ map translate . chop 9 $ V.toList hyp
            where
                translate = concatMap (\s -> toCharDebug s ++ " ")

printResult :: HypothesesSpace -> IO ()
printResult = putStrLn . toString
    where
        toString :: HypothesesSpace -> String
        toString hyp = translate (V.toList hyp)
            where
                translate = concatMap (\s -> toChar s ++ "")

-- The entire solution process!
solve :: String -> Maybe HypothesesSpace
solve str = do
    initialize str >>= prune >>= search

main :: IO ()
main = do
    [f] <- getArgs
    sudoku <- fmap lines $ readFile f -- "test.txt"
    mapM_ printResult $ mapMaybe solve sudoku
```

22 List comprehensions

by Ben Lynn.

Translated from my brute force solver in C (http://benlynn.blogspot.com/2012/04/sudoku-knuth-vs-norvig-vs-cohen.html) :

```
module Main where
f x s@(h:y)=let(r,c)=divMod(length x)9;m#n=m`div`3==n`div`3;e=[0..8]in
  [a|z<-['1'..'9'],h==z||h=='.'&&notElem z(map((x++s)!!)[i*9+j|i<-e,
  j<-e,i==r||j==c||i#r&&j#c]),a<-f(x++[z])y]
f x[]=[x]

main=print$f[] "53..7....6..195....98....6.8...6...34..8.3..17...2...6.6....28....419..5....8
```

23 Add your own

If you have a Sudoku solver you're proud of, put it here. This ought to be a good way of helping people learn some fun, intermediate-advanced techniques in Haskell.

24 Test boards

Here's an input file to test the solvers on. Zeroes represent blanks.

```
0 5 0 0 6 0 0 0 1
0 0 4 8 0 0 0 7 0
8 0 0 0 0 0 0 5 2
2 0 0 0 5 7 0 3 0
0 0 0 0 0 0 0 0 0
0 3 0 6 9 0 0 0 5
7 9 0 0 0 0 0 0 8
0 1 0 0 0 6 5 0 0
5 0 0 0 3 0 0 6 0
```

A nefarious one:

```
0 0 0 0 6 0 0 8 0
0 2 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 7 0 0 0 0 1 0 2
5 0 0 0 3 0 0 0 0
0 0 0 0 0 0 4 0 0
0 0 4 2 0 1 0 0 0
3 0 0 7 0 0 6 0 0
0 0 0 0 0 0 0 5 0
```

Chris Kuklewicz writes, "You can get over 47,000 distict minimal puzzles from csse.uwa.edu (http://www.csse.uwa.edu.au/~gordon/sudokumin.php) that have only 17 clues. Then you can run all of them through your program to locate the most evil ones, and use them on your associates."

Category:

- Code
