The Monad.Reader/Issue4/Solving Sudoku

From HaskellWiki

< The Monad.Reader | Issue4

This article needs reformatting! Please help tidy it up.--WouterSwierstra 14:23, 9 May 2008 (UTC)

Contents

1 Solving Sudoku

When I first started writing this article, Sudoku was - as far as I knew - the name of a number-placing puzzle of Japanese origin, set by The Times newspaper. Since then, almost every UK national newspaper has started carrying a weekly or even daily Sudoku puzzle; even the Northampton Chronicle has them (although the Chron's are usually pitifully easy). Sudoku has become a popular craze, and books of puzzles together with hints on how to solve them are even now on sale in newsagents up and down the country.

I started thinking about how to write an automated solver for Sudoku almost as soon as I was introduced to the Times' daily puzzle by my father. On the face of it, it's a fairly straightforward task. The Sudoku "board" is a grid of nine-by-nine squares, which is further subdivided into nine three-by-three cells. Some of the squares have a digit between 1 and 9 written in them; the objective of the puzzle is to place digits in the remaining squares such that every row, every column, and every three-by-three region contains every digit from 1 to 9. Each puzzle has only a single solution, which in the case of a properly-set puzzle can always be arrived at through a process of deduction (that is, without recourse to trial-and-error methods).

The solver I wrote, which is presented here, is not a purely deductive solver. I had never attempted to solve a Sudoku puzzle by hand when I came up with the algorithm, and was unaware of the full range of deductive procedures that are possible (there may still be

quite a few I haven't fathomed). My solver uses a combination of brute-force search, rule-based pruning of the search tree, and a simple but effective heuristic to guide it to the correct answer more quickly. It was written in Haskell because I wanted to explore some methods of writing programs of this kind in a purely functional language.

1.1 Grids and moves

Here is an example Sudoku puzzle:

|| # ||<#999999> A ||<#999999> B ||<#999999> C ||<#999999> D ||<#999999> E ||<#999999> F ||<#999999> G ||<#999999> H ||<#999999> I || ||<#999999> 1 ||<#EEEEEE> 7 ||<#EEEEEE> - ||<#EEEEEE> - || 1 || - || - ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> 2 || ||<#999999> 2 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || - || - || 6 ||<#EEEEEE> - ||<#EEEEEE> 8 ||<#EEEEEE> - || ||<#999999> 3 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || 8 || - || - ||<#EEEEEE> 1 ||<#EEEEEE> - ||<#EEEEEE> 9 || ||<#999999> 4 || - || - || 7 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> 9 || - || 1 || - || ||<#999999> 5 || - || 9 || 3 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || 5 || 4 || - || ||<#999999> 6 || - || 6 || - ||<#EEEEEE> 4 ||<#EEEEEE> - ||<#EEEEEE> - || 9 || - || - || ||<#999999> 7 ||<#EEEEEE> 3 ||<#EEEEEE> - ||<#EEEEEE> 8 || - || - || 4 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || ||<#999999> 8 ||<#EEEEEE> - ||<#EEEEEE> 4 ||<#EEEEEE> - || 3 || - || - ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || ||<#999999> 9 ||<#EEEEEE> 1 ||<#EEEEEE> - ||<#EEEEEE> - || - || - || 5 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> 3 ||

We can encode the puzzle as a list of positions together with the numbers that appear in each position, e.g.

[A1:7, D1:1, I1:2, F2:6, H2:8, D3:8, G3:1, I3:9, C4:7, F4:9, H4:1, B5:9, C5:3, G5:5, H5:4, B6:6, D6:4, G6:9, A7:3, C7:8, F7:4, B8:4, D8:3, A9:1, F9:5, I9:3]

Using the same notation, we can describe any new numbers that we write onto the grid; for instance, G9:4, which would give the grid:

|| # ||<#999999> A ||<#999999> B ||<#999999> C ||<#999999> D ||<#999999> E ||<#999999> F ||<#999999> G ||<#999999> H ||<#999999> I || ||<#999999> 1 ||<#EEEEEE> 7 ||<#EEEEEE> - ||<#EEEEEE> - || 1 || - || - ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> 2 || ||<#999999> 2 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || - || - || 6 ||<#EEEEEE> - ||<#EEEEEE> 8 ||<#EEEEEE> - || ||<#999999> 3 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || 8 || - || - ||<#EEEEEE> 1 ||<#EEEEEE> - ||<#EEEEEE> 9 || ||<#999999> 4 || - || - || 7 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> 9 || - || 1 || - || ||<#999999> 5 || - || 9 || 3 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || 5 || 4 || - || ||<#999999> 6 || - || 6 || - ||<#EEEEEE> 4 ||<#EEEEEE> - ||<#EEEEEE> - || 9 || - || - || ||<#999999> 7 ||<#EEEEEE> 3 ||<#EEEEEE> - ||<#EEEEEE> 8 || - || - || 4 ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || ||<#999999> 8 ||<#EEEEEE> - ||<#EEEEEE> 4 ||<#EEEEEE> - || 3 || - || - ||<#EEEEEE> - ||<#EEEEEE> - ||<#EEEEEE> - || ||<#999999> 9 ||<#EEEEEE> 1 ||<#EEEEEE> - ||<#EEEEEE> - || - || - || 5 ||<#FF4444> 4 ||<#EEEEEE> - ||<#EEEEEE> 3 ||

One way to think of this in Haskell is to think of each new number added to a specific cell in the grid as a "move", the state of the grid before the move as a value of some type,

and the state of the grid after the move as another value of the same type. The act of adding a number to the grid can then be represented as a function:

```
move :: SudokuState -> Move -> SudokuState
```

To set up a puzzle, given a list of initially-placed digits and a `SudokuState` representing a blank grid, we could then "fold" the list of initial moves into the grid like this:

```
moves :: SudokuState -> [Move] -> SudokuState
moves = foldl move
```

Given a function, Validate, that returns a value of True if a `SudokuState` is valid and False if it is not, we can already try any sequence of random moves and find out which ones result in a grid that breaks the rules of Sudoku and which ones result in a valid grid. The rules we have to test for are simply that no number may appear more than once in any given row, column, or 3x3 sub-grid.

1.2 Forced moves

Rather than trying moves that may break the rules and then testing the resulting grid to see whether they do or not, we can analyse the grid and find out which moves are permissible and which are not. One way to do this is to give every empty cell a list of all nine digits, and then remove from that list all of the digits that occur elsewhere in the same row, column or 3x3 sub-grid as that cell. If we are lucky, we may find that some cells have only one digit left in their list of possible digits after we have done this. These cells give rise to *forced* moves, moves that we have no alternative but to play.

If we can find all of the forced moves in a given position and play them, then we will arrive at a new position in which there may be other moves that are forced by the moves we have just played. It's possible to devise a Sudoku puzzle that could be completely solved just by doing this, but in fact most interesting puzzles require other kinds of deductive steps to reach a conclusion. Nevertheless, identifying and playing forced moves helps us to get closer to a solution without recourse to trial-and-error.

Just as playing forced moves may bring us nearer to a solution, it can also help us to identify a grid that *cannot* be solved, because we have placed the numbers in such a way that there are sequences of forced moves that result in an invalid grid. By using forced moves to look ahead for unavoidable failure, we can save ourselves the trouble of trying any more combinations of moves based on a hopeless position. Instead, we can go back immediately to a point where we had some choice about what move to play, and play a different move.

1.3 Monads: Definitely Maybe

Now, it happens that our algorithm involves two kinds of operation, transitions from one state to another and failing and backtracking to a previous state, that can be represented quite conveniently in Haskell using monads.

The `State` monad allows us to thread a state value invisibly through a sequence of actions that alter that state, so that instead of explicitly taking that value, applying a change to it and returning an altered value, we can express the state-changing actions in a more imperative style. We can thus replace

```
move :: SudokuState -> Move -> SudokuState
```

with

```
doMove :: Move -> SudokuState ()
```

where `SudokuState` is a version of the State monad that encapsulates changes to the state of a Sudoku grid. We can then rewrite

```
moves :: SudokuState -> [Move] -> SudokuState
moves = foldl move
```

as

```
doMoves :: [Move] -> SudokuState ()
doMoves = mapM_ doMove
```

The `State` monad also has some helper functions defined, `gets` and `modify`, that obtain a value from the encapsulated state using a projection function, and update it using a transformation function.

For failing and backtracking, we can use the `Maybe` monad, which is a member of the `MonadPlus` typeclass. The `MonadPlus` laws, applied to the `Maybe` monad, mean that

```
(Just a) `mplus` (Just b) = Just a
Nothing `mplus` (Just b) = Just b
(Just a) `mplus` Nothing = Just a
Nothing `mplus` Nothing = Nothing
```

In practice, this means that if we have two functions that return a value in the `Maybe` monad, we can `mplus` them together and take the first that succeeds (failure is defined as returning `Nothing`, which is done by `mzero`). Based on this, we can combine a series of possible moves using `msum`, and return the first that succeeds. This makes it possible to perform a depth-first traversal of a search tree, by summing together all of the branches at each node in the tree: the first branch that succeeds is taken.

To combine the `State` and `Maybe` monads, we use the monad transformer `StateT`. This allows us to return values in the `Maybe` monad from the `State` monad, and gives us the semantics of both monads combined in a useful way:

```
type StatePlus s a = StateT s Maybe a
```

(I am grateful to Michael Weber for showing me how to do this)

1.4 Implementing the Solver

This section is a literate Haskell implementation of the solver. We start with some external resources we're going to need.

```
> import List
> import Monad
> import Control.Monad.State
> import Control.Monad.Trans
```

Now for the basic types used to represent the contents of the Sudoku world.

The state of the grid can be represented simply as a list of lists of integers: a 9x9 array of cells. Empty cells are represented by a zero.

```
> type SudokuGrid = [[Int]]
> times n v = take n $ repeat v
> emptyRow :: [Int]
> emptyRow = times 9 0
> initialGrid = times 9 emptyRow
```

A position on the grid is represented by a co-ordinate pair of two integers, and a move is represented by a position and a digit.

```
> type Position = (Int, Int)
> type Move = (Position, Int)
```

The State/Maybe monad encapsulating the game state is defined using a monad transformer.

```
> type StatePlus s a = StateT s Maybe a
> type SudokuState a = StatePlus SudokuGrid a
```

We need some ways to read values out of the grid, and update its contents. For this we use `gets` and `modify`, supplying a variety of projection and transformation functions.

We also define three helper functions: `slice` and `select`, which are used to obtain ranges of values from the grid, and `replace`, which replaces a member of a list at a given index.

```
> getRows :: SudokuState [[Int]]
> getRows = gets id

> getRow :: Int -> SudokuState [Int]
> getRow index = getRows >>= return . flip (!!) index

> select :: [[Int]] -> [Int] -> [[Int]]
> select values range = map (\n -> values !! n) range
> slice :: [[Int]] -> [Int] -> [[Int]]
> slice values range = map (\n -> map (flip (!!) n) values) range
```

```
> getCols :: SudokuState [[Int]]
> getCols = gets $ \rows -> slice rows [0..8]

> getCol :: Int -> SudokuState [Int]
> getCol index = getCols >>= return . flip (!!) index

> getSubGrids :: SudokuState [[Int]]
> getSubGrids = gets $ \rows -> [concat $ slice (select rows [0..2]) [0..2],
> concat $ slice (select rows [0..2]) [3..5],
> concat $ slice (select rows [0..2]) [6..8],
> concat $ slice (select rows [3..5]) [0..2],
> concat $ slice (select rows [3..5]) [3..5],
> concat $ slice (select rows [3..5]) [6..8],
> concat $ slice (select rows [6..8]) [0..2],
> concat $ slice (select rows [6..8]) [3..5],
> concat $ slice (select rows [6..8]) [6..8]]

> getSubGrid :: Int -> SudokuState [Int]
> getSubGrid index = getSubGrids >>= return . flip (!!) index

> getCellValue :: Int -> Int -> SudokuState Int
> getCellValue x y = gets $ \rows -> (rows !! y) !! x

> getAllCellValues :: SudokuState [(Position, Int)]
> getAllCellValues = gets $ \rows ->
> let indexed = zip rows [0..] in
> concatMap (\(row, y) -> zip (zip [0..8] (times 9 y)) row) indexed

> replace index value list =
> let indexed = zip list [0..] in
> map (\(v, i) -> if i==index then value else v) indexed

> update :: Int -> Int -> Int -> SudokuState ()
> update x y v = modify $ \rows -> replace y (replace x v (rows !! y)) rows
```

Based on this, we can define `doMove` using `update`, and `doMoves` using `Move`.

```
> doMove ((x, y), v) = update x y v
> doMoves = mapM_ doMove
```

For notational convenience, we define a little parser for moves. `val` converts digit characters into Ints; `val'` does the same for the letters A-H.

```
> val c = let (Just p) = elemIndex c ['1'..'9'] in p
> val' c = let (Just p) = elemIndex c ['A'..'I'] in p
> makeMove (x:y:_:z:_) = ((val' x, val y), (val z) + 1)
```

To set up the initial game state, we apply a list of moves to the empty grid.

```
> initialize = doMoves . map makeMove
```

Now we can do some solving! The first thing we need to do is identify all of the empty squares in the grid. We then build a list of the permitted values in each of those squares.

(N.B. `containingSubGrid` returns the index of the subgrid containing a given position.)

```
> containingSubGrid (x, y) = (x `div` 3) + (3 * (y `div` 3))
```

```
> getEmptyPositions = do
> allCellValues <- getAllCellValues
> return $ map (\(p, _) -> p) . filter (\(_, v) -> v == 0) $ allCellValues

> getPermittedValues (x, y) = do
> row <- getRow y
> col <- getCol x
> subGrid <- getSubGrid $ containingSubGrid (x, y)
> return $ [1..9] \\ (row ++ col ++ subGrid)

> getPossibleMoves = do
> getEmptyPositions >>= mapM (\pos -> do pv <- getPermittedValues pos; return (pos, pv))
```

Once we have this analysis of what moves are possible, we can find all the forced moves in the current position. These are any positions where there is only one move permitted.

First we will find the forced moves and make them, then we will check that the current grid is valid. If it is, we will take one of the positions with the lowest number of possible moves, and try each of them in turn.

```
> atom (x:[]) = True
> atom _ = False

> getForcedMoves = do
> possibleMoves <- getPossibleMoves
> let forcedMoves = filter (\(_, ms) -> atom ms) possibleMoves
> return [(pos, v)|(pos, (v:_)) <- forcedMoves]

> doForcedMoves = do
> forcedMoves <- getForcedMoves
> if null forcedMoves
> then return ()
> else doMoves forcedMoves >> doForcedMoves

> getNextMove = do
> possibleMoves <- getPossibleMoves
> let blocked = filter (\(_, cs) -> null cs) possibleMoves
> if null possibleMoves
> then return []
> else if not (null blocked)
> then mzero
> else return (fewestOptions possibleMoves)

> fewestOptions es =
> let (e:_) = sortBy (\(_, cs1) (_, cs2) -> compare (length cs1) (length cs2)) es
> (pos, cs) = e in
> [(pos, c)|c <- cs]

> solve = do
> doForcedMoves
> valid <- checkValid
> if valid
> then do nextMove <- getNextMove
> if null nextMove
> then return ()
> else msum . map (\move -> doMove move >> solve) $ nextMove
> else mzero

> checkValid = do
> rows <- getRows
> cols <- getCols
> subGrids <- getSubGrids
> return $ (allValid rows) && (allValid cols) && (allValid subGrids)

> allValid = foldr (\x b -> b && (noDups x)) True
```

```
> noDups [] = True
> noDups (v:vs) = (if v/=0 then (not (v `elem` vs)) else True) && (noDups vs)
```

All that's left is to provide some pretty printing for results, and to create a `play` function that can invoke the solver.

```
> prettyPrint :: [[Int]] -> IO ()
> prettyPrint rows = mapM_ prettyPrintRow rows >> putStr "\n"

> prettyPrintRow r = mapM_ prettyPrintPos r >> putStr "\n"

> prettyPrintPos 0 = putStr "."
> prettyPrintPos p = putStr (show p)

> play moves = case (execStateT (initialize moves >> solve) initialGrid) of
> (Just result) -> prettyPrint result
> Nothing -> putStrLn "Failed"
```

For testing purposes, here's a sample puzzle and solution.

```
> sample = ["A1:1", "C2:2", "D2:7", "E2:4", "D3:5", "I3:4", "B4:3", "A5:7", "B5:5", "F6:9", "G6:6", "B7:4", "F7:

> solution = ["184963725",
> "562748319",
> "397512864",
> "239657148",
> "756184293",
> "418239657",
> "941376582",
> "623895471",
> "875421936"]

> solutionGrid = map (map (\c -> (val c) + 1)) solution

> test = case (execStateT (initialize sample >> solve) initialGrid) of
> (Just result) -> result==solutionGrid
> Nothing -> False
```

1.5 Conclusion

I hope that this program shows that it is easy to write simple recursive solvers in Haskell, with state and backtracking semantics layered into the program using monads and monad transformers. In terms of functional programming elegance, I feel this program is on the launch pad but has not yet taken off: the mechanism used for recursion should probably be made generic, and separated out from the particular program that uses it. There are other ways of practising non-determinism in Haskell, too: the List monad is typically used whenever multiple correct results are possible.

Finally, the algorithm presented here is neither particularly intelligent nor particularly fast: there are other ways of deducing forced moves, given an initial state, and all well-written Sudoku puzzles should be able to be solved using only deduction (which would eliminate the need for backtracking altogether).

In practice, it is much more rewarding to solve Sudoku puzzles manually than to feed them into a computer and wait a few seconds for an automated solution. On the other hand, writing a solver such as this one can also be a stimulating challenge, and I hope

you have enjoyed reading about it

Retrieved from "http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue4/Solving_Sudoku"