Fast Sudoku Solver in Haskell #3: Picking the Right Data Structures

August 13, 2018
A thirty-four minute read
Tags: haskell, sudoku, programming, puzzle, nilenso

In the previous part in this series of posts, we optimized the simple Sudoku solver by implementing a new strategy to prune cells, and were able to achieve a speedup of almost 200x. Afterwards, we profiled the solution and found that there were bottlenecks in the program, leading to a slowdown. In this post, we are going to follow the profiler and use the right *Data Structures* to improve the solution further and make it **faster**.

This is the third post in a series of posts:

1. Fast Sudoku Solver in Haskell #1: A Simple Solution
2. Fast Sudoku Solver in Haskell #2: A 200x Faster Solution
3. Fast Sudoku Solver in Haskell #3: Picking the Right Data Structures

Discuss this post on r/haskell.

Contents

Quick Recap

Sudoku is a number placement puzzle. It consists of a 9x9 grid which is to be filled with digits from 1 to 9 such that each row, each column and each of the nine 3x3 sub-grids contain all the digits. Some of the cells of the grid come pre-filled and the player has to fill the rest.

In the previous post, we improved the performance of the simple Sudoku solver by implementing a new strategy to prune cells. This new strategy found the digits which occurred uniquely, in pairs, or in triplets and fixed the cells to those digits. It led to a speedup of about 200x over our original naive solution. This is our current run[1] time for solving all the 49151 17-clue puzzles:

```
$ cat sudoku17.txt | time stack exec sudoku > /dev/null
      258.97 real       257.34 user         1.52 sys
```

Let's try to improve this time.[2]

Profile Twice, Code Once

Instead of trying to guess how to improve the performance of our solution, let's be methodical about it. We start with profiling the code to find the bottlenecks. Let's compile and run the code with profiling flags:

```
$ stack build --profile
$ head -1000 sudoku17.txt | stack exec -- sudoku +RTS -p > /dev/null
```

This generates a `sudoku.prof` file with the profiling output. Here are the top seven *Cost Centres*[3] from the file (cleaned for brevity):

Cost Centre	Src	%time	%alloc
`exclusivePossibilities`	Sudoku.hs:(49,1)-(62,26)	18.9	11.4
`pruneCellsByFixed.pruneCell`	Sudoku.hs:(75,5)-(76,36)	17.7	30.8
`exclusivePossibilities.\.\`	Sudoku.hs:55:38-70	11.7	20.3
`fixM.\`	Sudoku.hs:13:27-65	10.7	0.0
`==`	Sudoku.hs:15:56-57	5.6	0.0
`pruneGrid'`	Sudoku.hs:(103,1)-(106,64)	5.0	6.7
`pruneCellsByFixed`	Sudoku.hs:(71,1)-(76,36)	4.5	5.0
`exclusivePossibilities.\`	Sudoku.hs:58:36-68	3.4	2.5

Cost Centre points to a function, either named or anonymous. *Src* gives the line and column numbers of the source code of the function. *%time* and *%alloc* are the percentages of time spent and memory allocated in the function, respectively.

We see that `exclusivePossibilities` and the nested functions inside it take up almost 34% time of the entire run time. Second biggest bottleneck is the `pruneCell` function inside the `pruneCellsByFixed` function.

We are going to look at `exclusivePossibilities` later. For now, it is easy to guess the possible reason for `pruneCell` taking so much time. Here's the code for reference:

```
1 pruneCellsByFixed :: [Cell] -> Maybe [Cell]
2 pruneCellsByFixed cells = traverse pruneCell cells
3   where
4     fixeds = [x | Fixed x <- cells]
5
6     pruneCell (Possible xs) = makeCell (xs Data.List.\\ fixeds)
7     pruneCell x             = Just x
```

`pruneCell` uses `Data.List.\\` to find the difference of the cell's possible digits and the fixed digits in the cell's block. In Haskell, lists are implemented as singly linked lists. So, finding the difference or intersection of two lists is $O(n^2)$, that is, quadratic asymptotic complexity. Let's tackle this bottleneck first.

A Set for All Occasions

What is a efficient data structure for finding differences and intersections? Why, a *Set* of course! A Set stores unique values and provides fast operations for testing membership of its elements. If we use a Set to represent the possible values of cells instead of a List, the program should run faster. Since the possible values are already unique (`1`–`9`), it should not break anything.

Haskell comes with a bunch of Set implementations:

- `Data.Set` which is a generic data structure implemented as self-balancing binary search tree.
- `Data.HashSet` which is a generic data structure implemented as hash array mapped trie.
- `Data.IntSet` which is a specialized data structure for integer values, implemented as radix tree.

However, a much faster implementation is possible for our particular use-case. We can use a *BitSet*.

A BitSet uses bits to represent unique members of a Set. We map values to particular bits using some function. If the bit corresponding to a particular value is set to 1 then the value is present in the Set, else it is not. So, we need as many bits in a BitSet as the number of values in our domain, which makes is difficult to use for generic problems. But, for our Sudoku solver, we need to store only the digits `1`–`9` in the Set, which make BitSet very suitable for us. Also, the Set operations on BitSet are implemented using bit-level instructions in hardware, making them much faster than those on the other data structure listed above.

In Haskell, we can use the `Data.Word` module to represent a BitSet. Specifically, we can use the `Data.Word.Word16` type which has sixteen bits because we need only nine bits to represent the nine digits. The bit-level operations on `Word16` are provided by the `Data.Bits` module.

Bit by Bit, We Get Faster

First, we replace List with `Word16` in the `Cell` type and add a helper function:

```
1  data Cell = Fixed Data.Word.Word16
2            | Possible Data.Word.Word16
3            deriving (Show, Eq)
4
5  setBits :: Data.Word.Word16 -> [Data.Word.Word16] -> Data.Word.Word16
6  setBits = Data.List.foldl' (Data.Bits..|.)
```

Then we replace `Int` related operations with bit related ones in the read and show functions:

```haskell
readGrid :: String -> Maybe Grid
readGrid s
  | length s == 81 =
      traverse (traverse readCell) . Data.List.Split.chunksOf 9 $ s
  | otherwise      = Nothing
  where
    allBitsSet = 1022

    readCell '.' = Just $ Possible allBitsSet
    readCell c
      | Data.Char.isDigit c && c > '0' =
          Just . Fixed . Data.Bits.bit . Data.Char.digitToInt $ c
      | otherwise = Nothing

showGrid :: Grid -> String
showGrid = unlines . map (unwords . map showCell)
  where
    showCell (Fixed x) = show . Data.Bits.countTrailingZeros $ x
    showCell _         = "."

showGridWithPossibilities :: Grid -> String
showGridWithPossibilities = unlines . map (unwords . map showCell)
  where
    showCell (Fixed x) = (show . Data.Bits.countTrailingZeros $ x) ++ "          "
    showCell (Possible xs) =
      "[" ++
      map (\i -> if Data.Bits.testBit xs i
                 then Data.Char.intToDigit i
                 else ' ')
          [1..9]
      ++ "]"
```

We set the same bits as the digits to indicate the presence of the digits in the possibilities. For example, for digit `1`, we set the bit 1 so that the resulting `Word16` is `0000 0000 0000 0010` or 2. This also means, for fixed cells, the value is count of the zeros from right.

The change in the `exclusivePossibilities` function is pretty minimal:

```
-exclusivePossibilities :: [Cell] -> [[Int]]
+exclusivePossibilities :: [Cell] -> [Data.Word.Word16]
 exclusivePossibilities row =
   row
   & zip [1..9]
   & filter (isPossible . snd)
   & Data.List.foldl'
       (\acc ~(i, Possible xs) ->
-        Data.List.foldl'
-          (\acc' x -> Map.insertWith prepend x [i] acc')
-          acc
-          xs)
+        Data.List.foldl'
+          (\acc' x -> if Data.Bits.testBit xs x
+                      then Map.insertWith prepend x [i] acc'
+                      else acc')
+          acc
+          [1..9])
       Map.empty
   & Map.filter ((< 4) . length)
   & Map.foldlWithKey' (\acc x is -> Map.insertWith prepend is [x] acc) Map.empty
   & Map.filterWithKey (\is xs -> length is == length xs)
   & Map.elems
+  & map (Data.List.foldl' Data.Bits.setBit Data.Bits.zeroBits)
   where
     prepend ~[y] ys = y:ys
```

In the nested folding step, instead of folding over the possible values of the cells, now we fold over the digits from `1` to `9` and insert the entry in the map if the bit corresponding to the digit is set in the possibilities. And as the last step, we convert the exclusive possibilities to `Word16` by folding them, starting with zero. As example in the *REPL* should be instructive:

```
*Main> poss = Data.List.foldl' Data.Bits.setBit Data.Bits.zeroBits
*Main> row = [Possible $ poss [4,6,9], Fixed $ poss [1], Fixed $ poss [5], Possible $
poss [6,9], Fixed $ poss [7], Possible $ poss [2,3,6,8,9], Possible $ poss [6,9],
Possible $ poss [2,3,6,8,9], Possible $ poss [2,3,6,8,9]]
*Main> putStr $ showGridWithPossibilities [row]
[   4 6  9] 1           5           [     6  9] 7           [ 23  6 89] [     6  9] [ 23
6 89] [ 23  6 89]
*Main> exclusivePossibilities row
[16,268]
*Main> [poss [4], poss [8,3,2]]
[16,268]
```

This is the same example row as the last time. And it returns same results, excepts as a list of `Word16` now.

Now, we change `makeCell` to use bit operations instead of list ones:

```
makeCell :: Data.Word.Word16 -> Maybe Cell
makeCell ys
  | ys == Data.Bits.zeroBits   = Nothing
  | Data.Bits.popCount ys == 1 = Just $ Fixed ys
  | otherwise                  = Just $ Possible ys
```

And we change cell pruning functions too:

```
 pruneCellsByFixed :: [Cell] -> Maybe [Cell]
 pruneCellsByFixed cells = traverse pruneCell cells
   where
-    fixeds = [x | Fixed x <- cells]
+    fixeds = setBits Data.Bits.zeroBits [x | Fixed x <- cells]

-    pruneCell (Possible xs) = makeCell (xs Data.List.\\ fixeds)
+    pruneCell (Possible xs) =
+      makeCell (xs Data.Bits..&. Data.Bits.complement fixeds)
     pruneCell x             = Just x

 pruneCellsByExclusives :: [Cell] -> Maybe [Cell]
 pruneCellsByExclusives cells = case exclusives of
   [] -> Just cells
   _  -> traverse pruneCell cells
   where
     exclusives    = exclusivePossibilities cells
-    allExclusives = concat exclusives
+    allExclusives = setBits Data.Bits.zeroBits exclusives

     pruneCell cell@(Fixed _) = Just cell
     pruneCell cell@(Possible xs)
       | intersection `elem` exclusives = makeCell intersection
       | otherwise                      = Just cell
       where
-        intersection = xs `Data.List.intersect` allExclusives
+        intersection = xs Data.Bits..&. allExclusives
```

Notice how the list difference and intersection functions are replaced by `Data.Bits` functions. Specifically, list difference is replace by bitwise-and of the bitwise-complement, and list intersection is replaced by bitwise-and.

We make a one-line change in the `isGridInvalid` function to find empty possible cells using bit ops:

```
isGridInvalid :: Grid -> Bool
isGridInvalid grid =
  any isInvalidRow grid
  || any isInvalidRow (Data.List.transpose grid)
  || any isInvalidRow (subGridsToRows grid)
  where
    isInvalidRow row =
      let fixeds         = [x | Fixed x <- row]
-         emptyPossibles = [x | Possible x <- row, null x]
+         emptyPossibles = [() | Possible x <- row, x == Data.Bits.zeroBits]
      in hasDups fixeds || not (null emptyPossibles)

    hasDups l = hasDups' l []

    hasDups' [] _ = False
    hasDups' (y:ys) xs
      | y `elem` xs = True
      | otherwise   = hasDups' ys (y:xs)
```

And finally, we change the `nextGrids` functions to use bit operations:

```
nextGrids :: Grid -> (Grid, Grid)
nextGrids grid =
  let (i, first@(Fixed _), rest) =
        fixCell
        . Data.List.minimumBy (compare `Data.Function.on` (possibilityCount . snd))
        . filter (isPossible . snd)
        . zip [0..]
        . concat
        $ grid
  in (replace2D i first grid, replace2D i rest grid)
  where
    possibilityCount (Possible xs) = Data.Bits.popCount xs
    possibilityCount (Fixed _)     = 1

    fixCell ~(i, Possible xs) =
      let x = Data.Bits.countTrailingZeros xs
      in case makeCell (Data.Bits.clearBit xs x) of
        Nothing -> error "Impossible case"
        Just cell -> (i, Fixed (Data.Bits.bit x), cell)

    replace2D :: Int -> a -> [[a]] -> [[a]]
    replace2D i v =
      let (x, y) = (i `quot` 9, i `mod` 9) in replace x (replace y (const v))
    replace p f xs = [if i == p then f x else x | (x, i) <- zip xs [0..]]
```

`possibilityCount` now uses `Data.Bits.popCount` to count the number of bits set to 1. `fixCell` now chooses the first set bit from right as the digit to fix. Rest of the code stays the same. Let's build and run it:

```
$ stack build
$ cat sudoku17.txt | time stack exec sudoku > /dev/null
       69.44 real        69.12 user         0.37 sys
```

Wow! That is almost 3.7x faster than the previous solution. It's a massive win! But let's not be content yet. To the profiler again![4]

Back to the Profiler

Running the profiler again gives us these top six culprits:

Cost Centre	Src	%time	%alloc
`exclusivePossibilities`	Sudoku.hs:(57,1)-(74,26)	25.2	16.6
`exclusivePossibilities.\.\`	Sudoku.hs:64:23-96	19.0	32.8
`fixM.\`	Sudoku.hs:15:27-65	12.5	0.1
`pruneCellsByFixed`	Sudoku.hs:(83,1)-(88,36)	5.9	7.1
`pruneGrid'`	Sudoku.hs:(115,1)-(118,64)	5.0	8.6

Hurray! `pruneCellsByFixed.pruneCell` has disappeared from the list of top bottlenecks. Though `exclusivePossibilities` still remains here as expected.

`exclusivePossibilities` is a big function. The profiler does not really tell us which parts of it are the slow ones. That's because by default, the profiler only considers functions as *Cost Centres*. We need to give it hints for it to be able to find bottlenecks inside functions. For that, we need to insert *Cost Centre* annotations in the code:

```haskell
exclusivePossibilities :: [Cell] -> [Data.Word.Word16]
exclusivePossibilities row =
  row
  & ({-# SCC "EP.zip" #-} zip [1..9])
  & ({-# SCC "EP.filter" #-} filter (isPossible . snd))
  & ({-# SCC "EP.foldl" #-} Data.List.foldl'
      (\acc ~(i, Possible xs) ->
        Data.List.foldl'
          (\acc' n -> if Data.Bits.testBit xs n
                      then Map.insertWith prepend n [i] acc'
                      else acc')
          acc
          [1..9])
      Map.empty)
  & ({-# SCC "EP.Map.filter1" #-} Map.filter ((< 4) . length))
  & ({-# SCC "EP.Map.foldl" #-}
       Map.foldlWithKey'
         (\acc x is -> Map.insertWith prepend is [x] acc)
         Map.empty)
  & ({-# SCC "EP.Map.filter2" #-}
       Map.filterWithKey (\is xs -> length is == length xs))
  & ({-# SCC "EP.Map.elems" #-} Map.elems)
  & ({-# SCC "EP.map" #-}
       map (Data.List.foldl' Data.Bits.setBit Data.Bits.zeroBits))
  where
    prepend ~[y] ys = y:ys
```

Here, `{-# SCC "EP.zip" #-}` is a *Cost Centre* annotation. `"EP.zip"` is the name we choose to give to this *Cost Centre*.

After profiling the code again, we get a different list of bottlenecks:

Cost Centre	Src	%time	%alloc
`exclusivePossibilities.\.\`	Sudoku.hs:(64,23)-(66,31)	19.5	31.4
`fixM.\`	Sudoku.hs:15:27-65	13.1	0.1
`pruneCellsByFixed`	Sudoku.hs:(85,1)-(90,36)	5.4	6.8
`pruneGrid'`	Sudoku.hs:(117,1)-(120,64)	4.8	8.3
`EP.zip`	Sudoku.hs:59:27-36	4.3	10.7
`EP.Map.filter1`	Sudoku.hs:70:35-61	4.2	0.5
`chunksOf`	Data/List/Split/Internals.hs:(514,1)-(517,49)	4.1	7.4
`exclusivePossibilities.\`	Sudoku.hs:71:64-96	4.0	3.4
`EP.filter`	Sudoku.hs:60:30-54	2.9	3.4
`EP.foldl`	Sudoku.hs:(61,29)-(69,15)	2.8	1.8
`exclusivePossibilities`	Sudoku.hs:(57,1)-(76,26)	2.7	1.9
`chunksOf.splitter`	Data/List/Split/Internals.hs:(516,3)-(517,49)	2.5	2.7

So almost one-fifth of the time is actually going in this nested one-line anonymous function inside `exclusivePossibilities`:

```
(\acc' n ->
    if Data.Bits.testBit xs n then Map.insertWith prepend n [i] acc' else acc')
```

But we are going to ignore it for now.

If we look closely, we also find that around 17% of the run time now goes into list traversal and manipulation. This is in the functions `pruneCellsByFixed`, `pruneGrid'`, `chunksOf` and `chunksOf.splitter`, where the first two are majorly list traversal and transposition, and the last two are list splitting. Maybe it is time to get rid of lists altogether?

Vectors of Speed

Vector is a Haskell library for working with arrays. It implements very performant operations for integer-indexed array data. Unlike the lists in Haskell which are implemented as singly linked lists, vectors are stored in a contiguous set of memory locations. This makes random access to the elements a constant time operation. The memory overhead per additional item in vectors is also much smaller. Lists allocate memory for each item in the heap and have pointers to the memory locations in nodes, leading to a lot of wasted memory in holding pointers. On the other hand, operations on lists are lazy, whereas, operations on vectors are strict, and this may need to useless computation depending on the use-case[5].

In our current code, we represent the grid as a list of lists of cells. All the pruning operations require us to traverse the grid list or the row lists. We also need to transform the grid back-and-forth for being able to use the same pruning operations for rows, columns and sub-grids. The pruning of cells and the choosing of pivot cells also requires us to replace cells in the grid with new ones, leading to a lot of list traversals.

To prevent all this linear-time list traversals, we can replace the nested list of lists with a single vector. Then all we need to do it to go over the right parts of this vector, looking up and replacing cells as needed. Since both lookups and updates on vectors are constant time, this should lead to a speedup.

Let’s start by changing the grid to a vector of cells.:

```
data Cell = Fixed Data.Word.Word16
          | Possible Data.Word.Word16
          deriving (Show, Eq)

type Grid = Data.Vector.Vector Cell
```

Since we plan to traverse different parts of the same vector, let’s define these different parts first:

```
type CellIxs = [Int]

fromXY :: (Int, Int) -> Int
fromXY (x, y) = x * 9 + y

allRowIxs, allColIxs, allSubGridIxs :: [CellIxs]
allRowIxs = [getRow i | i <- [0..8]]
  where getRow n = [ fromXY (n, i) | i <- [0..8] ]

allColIxs = [getCol i | i <- [0..8]]
  where getCol n = [ fromXY (i, n) | i <- [0..8] ]

allSubGridIxs = [getSubGrid i | i <- [0..8]]
  where getSubGrid n = let (r, c) = (n `quot` 3, n `mod` 3)
          in [ fromXY (3 * r + i, 3 * c + j) | i <- [0..2], j <- [0..2] ]
```

We define a type for cell indices as a list of integers. Then we create three lists of cell indices: all row indices, all column indices, and all sub-grid indices. Let's check these out in the *REPL*:

```
*Main> Control.Monad.mapM_ print allRowIxs
[0,1,2,3,4,5,6,7,8]
[9,10,11,12,13,14,15,16,17]
[18,19,20,21,22,23,24,25,26]
[27,28,29,30,31,32,33,34,35]
[36,37,38,39,40,41,42,43,44]
[45,46,47,48,49,50,51,52,53]
[54,55,56,57,58,59,60,61,62]
[63,64,65,66,67,68,69,70,71]
[72,73,74,75,76,77,78,79,80]
*Main> Control.Monad.mapM_ print allColIxs
[0,9,18,27,36,45,54,63,72]
[1,10,19,28,37,46,55,64,73]
[2,11,20,29,38,47,56,65,74]
[3,12,21,30,39,48,57,66,75]
[4,13,22,31,40,49,58,67,76]
[5,14,23,32,41,50,59,68,77]
[6,15,24,33,42,51,60,69,78]
[7,16,25,34,43,52,61,70,79]
[8,17,26,35,44,53,62,71,80]
*Main> Control.Monad.mapM_ print allSubGridIxs
[0,1,2,9,10,11,18,19,20]
[3,4,5,12,13,14,21,22,23]
[6,7,8,15,16,17,24,25,26]
[27,28,29,36,37,38,45,46,47]
[30,31,32,39,40,41,48,49,50]
[33,34,35,42,43,44,51,52,53]
[54,55,56,63,64,65,72,73,74]
[57,58,59,66,67,68,75,76,77]
[60,61,62,69,70,71,78,79,80]
```

We can verify manually that these indices are correct.

Read and show functions are easy to change for vector:

```
 readGrid :: String -> Maybe Grid
 readGrid s
-  | length s == 81 = traverse (traverse readCell) . Data.List.Split.chunksOf 9 $ s
+  | length s == 81 = Data.Vector.fromList <$> traverse readCell s
   | otherwise      = Nothing
   where
     allBitsSet = 1022

     readCell '.' = Just $ Possible allBitsSet
     readCell c
       | Data.Char.isDigit c && c > '0' =
           Just . Fixed . Data.Bits.bit . Data.Char.digitToInt $ c
       | otherwise = Nothing

 showGrid :: Grid -> String
-showGrid = unlines . map (unwords . map showCell)
+showGrid grid =
+  unlines . map (unwords . map (showCell . (grid !))) $ allRowIxs
   where
     showCell (Fixed x) = show . Data.Bits.countTrailingZeros $ x
     showCell _         = "."

 showGridWithPossibilities :: Grid -> String
-showGridWithPossibilities = unlines . map (unwords . map showCell)
+showGridWithPossibilities grid =
+  unlines . map (unwords . map (showCell . (grid !))) $ allRowIxs
   where
     showCell (Fixed x) = (show . Data.Bits.countTrailingZeros $ x) ++ "          "
     showCell (Possible xs) =
       "[" ++
       map (\i -> if Data.Bits.testBit xs i
                  then Data.Char.intToDigit i
                  else ' ')
           [1..9]
       ++ "]"
```

`readGrid` simply changes to work on a single vector of cells instead of a list of lists. Show functions have a pretty minor change to do lookups from a vector using the row indices and the `(!)` function. The `(!)` function is the vector indexing function which is similar to the `(!!)` function, except it executes in constant time.

The pruning related functions are rewritten for working with vectors:

```haskell
replaceCell :: Int -> Cell -> Grid -> Grid
replaceCell i c g = g Data.Vector.// [(i, c)]

pruneCellsByFixed :: Grid -> CellIxs -> Maybe Grid
pruneCellsByFixed grid cellIxs =
  Control.Monad.foldM pruneCell grid . map (\i -> (i, grid ! i)) $ cellIxs
  where
    fixeds = setBits Data.Bits.zeroBits [x | Fixed x <- map (grid !) cellIxs]

    pruneCell g (_, Fixed _) = Just g
    pruneCell g (i, Possible xs)
      | xs' == xs = Just g
      | otherwise = flip (replaceCell i) g <$> makeCell xs'
      where
        xs' = xs Data.Bits..&. Data.Bits.complement fixeds

pruneCellsByExclusives :: Grid -> CellIxs -> Maybe Grid
pruneCellsByExclusives grid cellIxs = case exclusives of
  [] -> Just grid
  _  -> Control.Monad.foldM pruneCell grid . zip cellIxs $ cells
  where
    cells         = map (grid !) cellIxs
    exclusives    = exclusivePossibilities cells
    allExclusives = setBits Data.Bits.zeroBits exclusives

    pruneCell g (_, Fixed _) = Just g
    pruneCell g (i, Possible xs)
      | intersection == xs             = Just g
      | intersection `elem` exclusives =
          flip (replaceCell i) g <$> makeCell intersection
      | otherwise                      = Just g
      where
        intersection = xs Data.Bits..&. allExclusives

pruneCells :: Grid -> CellIxs -> Maybe Grid
pruneCells grid cellIxs =
  fixM (flip pruneCellsByFixed cellIxs) grid
  >>= fixM (flip pruneCellsByExclusives cellIxs)
```

All the three functions now take the grid and the cell indices instead of a list of cells, and use the cell indices to lookup the cells from the grid. Also, instead of using the `traverse` function as earlier, now we use the `Control.Monad.foldM` function to fold over the cell-index-and-cell tuples in the context of the `Maybe` monad, making changes to the grid directly.

We use the `replaceCell` function to replace cells at an index in the grid. It is a simple wrapper over the vector update function `Data.Vector.//`. Rest of the code is same in essence, except a few changes to accommodate the changed function parameters.

`pruneGrid'` function does not need to do transpositions and back-transpositions anymore as now we use the cell indices to go over the right parts of the grid vector directly:

```
pruneGrid' :: Grid -> Maybe Grid
pruneGrid' grid =
  Control.Monad.foldM pruneCells grid allRowIxs
  >>= flip (Control.Monad.foldM pruneCells) allColIxs
  >>= flip (Control.Monad.foldM pruneCells) allSubGridIxs
```

Notice that the `traverse` function here is also replaced by the `Control.Monad.foldM` function.

Similarly, the grid predicate functions change a little to go over a vector instead of a list of lists:

```
  isGridFilled :: Grid -> Bool
-isGridFilled grid = null [ () | Possible _ <- concat grid ]
+isGridFilled = not . Data.Vector.any isPossible

  isGridInvalid :: Grid -> Bool
  isGridInvalid grid =
-   any isInvalidRow grid
-   || any isInvalidRow (Data.List.transpose grid)
-   || any isInvalidRow (subGridsToRows grid)
+   any isInvalidRow (map (map (grid !)) allRowIxs)
+   || any isInvalidRow (map (map (grid !)) allColIxs)
+   || any isInvalidRow (map (map (grid !)) allSubGridIxs)
```

And finally, we change the `nextGrids` function to replace the list related operations with the vector related ones:

```
  nextGrids :: Grid -> (Grid, Grid)
  nextGrids grid =
    let (i, first@(Fixed _), rest) =
          fixCell
-         . Data.List.minimumBy
+         . Data.Vector.minimumBy
              (compare `Data.Function.on` (possibilityCount . snd))
-         . filter (isPossible . snd)
-         . zip [0..]
-         . concat
+         . Data.Vector.imapMaybe
+             (\j cell -> if isPossible cell then Just (j, cell) else Nothing)
          $ grid
-  in (replace2D i first grid, replace2D i rest grid)
+  in (replaceCell i first grid, replaceCell i rest grid)
```

We also switch the `replace2D` function which went over the entire list of lists of cells to replace a cell, with the vector-based `replaceCell` function.

All the required changes are done. Let’s do a run:

```
$ stack build
$ cat sudoku17.txt | time stack exec sudoku > /dev/null
       88.53 real        88.16 user         0.41 sys
```

Oops! Instead of getting a speedup, our vector-based code is actually 1.3x slower than the list-based code. How did this happen? Time to bust out the profiler again!

Revenge of the `(==)`

Profiling the current code gives us the following hotspots:

Cost Centre	Src	%time	%alloc
`>>=`	Data/Vector/Fusion/Util.hs:36:3-18	52.2	51.0
`basicUnsafeIndexM`	Data/Vector.hs:278:3-62	22.2	20.4
`exclusivePossibilities`	Sudoku.hs:(75,1)-(93,26)	6.8	8.3
`exclusivePossibilities.\.\`	Sudoku.hs:83:23-96	3.8	8.8
`pruneCellsByFixed.fixeds`	Sudoku.hs:105:5-77	2.0	1.7

We see a sudden appearance of `(>>=)` from the `Data.Vector.Fusion.Util` module at the top of the list, taking more than half of the run time. For more clues, we dive into the detailed profiler report and find this bit:

Cost Centre	Src	%time	%alloc
`pruneGrid`	Sudoku.hs:143:1-27	0.0	0.0
`fixM`	Sudoku.hs:16:1-65	0.1	0.0
`fixM.\`	Sudoku.hs:16:27-65	0.2	0.1
`==`	Data/Vector.hs:287:3-50	1.0	1.4
`>>=`	Data/Vector/Fusion/Util.hs:36:3-18	51.9	50.7
`basicUnsafeIndexM`	Data/Vector.hs:278:3-62	19.3	20.3

Here, the indentation indicated nesting of operations. We see that both the `(>>=)` and `basicUnsafeIndexM` functions — which together take around three-quarter of the run time — are being called from the `(==)` function in the `fixM` function[6]. It seems like we are checking for equality too many times. Here's the usage of the `fixM` for reference:

```haskell
pruneCells :: Grid -> CellIxs -> Maybe Grid
pruneCells grid cellIxs =
  fixM (flip pruneCellsByFixed cellIxs) grid
  >>= fixM (flip pruneCellsByExclusives cellIxs)

pruneGrid :: Grid -> Maybe Grid
pruneGrid = fixM pruneGrid'
```

In `pruneGrid`, we run `pruneGrid'` till the resultant grid settles, that is, the grid computed in a particular iteration is **equal to** the grid in the previous iteration. Interestingly, we do the same thing in `pruneCells` too. We equate **the whole grid** to check for settling of each block of cells. This is the reason of the slowdown.

One Function to Prune Them All

Why did we add `fixM` in the `pruneCells` function at all? Quoting from the previous post,

> We need to run `pruneCellsByFixed` and `pruneCellsByExclusives` repeatedly using `fixM` because an unsettled row can lead to wrong solutions.
>
> Imagine a row which just got a `9` fixed because of `pruneCellsByFixed`. If we don't run the function again, the row may be left with one non-fixed cell with a `9`. When we run this row through `pruneCellsByExclusives`, it'll consider the `9` in the non-fixed cell as a *Single* and fix it. This will lead to two `9`s in the same row, causing the solution to fail.

So the reason we added `fixM` is that, we run the two pruning strategies one-after-another. That way, they see the cells in the same block in different states. If we were to merge the two pruning functions into a single one such that they work in lockstep, we would not need to run `fixM` at all!

With this idea, we rewrite `pruneCells` as a single function:

```haskell
pruneCells :: Grid -> CellIxs -> Maybe Grid
pruneCells grid cellIxs = Control.Monad.foldM pruneCell grid cellIxs
  where
    cells         = map (grid !) cellIxs
    exclusives    = exclusivePossibilities cells
    allExclusives = setBits Data.Bits.zeroBits exclusives
    fixeds        = setBits Data.Bits.zeroBits [x | Fixed x <- cells]

    pruneCell g i =
      pruneCellByFixed g (i, g ! i) >>= \g' -> pruneCellByExclusives g' (i, g' ! i)

    pruneCellByFixed g (_, Fixed _) = Just g
    pruneCellByFixed g (i, Possible xs)
      | xs' == xs = Just g
      | otherwise = flip (replaceCell i) g <$> makeCell xs'
      where
        xs' = xs Data.Bits..&. Data.Bits.complement fixeds

    pruneCellByExclusives g (_, Fixed _) = Just g
    pruneCellByExclusives g (i, Possible xs)
      | null exclusives               = Just g
      | intersection == xs            = Just g
      | intersection `elem` exclusives =
          flip (replaceCell i) g <$> makeCell intersection
      | otherwise                     = Just g
      where
        intersection = xs Data.Bits..&. allExclusives
```

We have merged the two pruning functions almost blindly. The important part here is the nested `pruneCell` function which uses monadic bind `(>>=)` to ensure that cells fixed in the first step are seen by the next step. Merging the two functions ensures that both strategies will see same *Exclusives* and *Fixeds*, thereby running in lockstep.

Let's try it out:

```
$ stack build
$ cat sudoku17.txt | time stack exec sudoku > /dev/null
      57.67 real        57.12 user         0.46 sys
```

Ah, now it's faster than the list-based implementation by 1.2x[7]. Let's see what the profiler says:

Cost Centre	Src	%time	%alloc
`exclusivePossibilities.\.\`	Sudoku.hs:82:23-96	15.7	33.3
`pruneCells`	Sudoku.hs:(101,1)-(126,53)	9.6	6.8
`pruneCells.pruneCell`	Sudoku.hs:(108,5)-(109,83)	9.5	2.1
`basicUnsafeIndexM`	Data/Vector.hs:278:3-62	9.4	0.5
`pruneCells.pruneCell.\`	Sudoku.hs:109:48-83	7.6	2.1
`pruneCells.cells`	Sudoku.hs:103:5-40	7.1	10.9
`exclusivePossibilities.\`	Sudoku.hs:87:64-96	3.5	3.8
`EP.Map.filter1`	Sudoku.hs:86:35-61	3.0	0.6
`>>=`	Data/Vector/Fusion/Util.hs:36:3-18	2.8	2.0
`replaceCell`	Sudoku.hs:59:1-45	2.5	1.1
`EP.filter`	Sudoku.hs:78:30-54	2.4	3.3
`primitive`	Control/Monad/Primitive.hs:195:3-16	2.3	6.5

The double nested anonymous function mentioned before is still the biggest culprit but `fixM` has disappeared from the list. Let's tackle `exclusivePossibilities` now.

Rise of the Mutables

Here's `exclusivePossibilities` again for reference:

```
1  exclusivePossibilities :: [Cell] -> [Data.Word.Word16]
2  exclusivePossibilities row =
3    row
4    & zip [1..9]
5    & filter (isPossible . snd)
6    & Data.List.foldl'
7        (\acc ~(i, Possible xs) ->
8          Data.List.foldl'
9            (\acc' n -> if Data.Bits.testBit xs n
10                       then Map.insertWith prepend n [i] acc'
11                       else acc')
12           acc
13           [1..9])
14       Map.empty
15   & Map.filter ((< 4) . length)
16   & Map.foldlWithKey'(\acc x is -> Map.insertWith prepend is [x] acc) Map.empty
17   & Map.filterWithKey (\is xs -> length is == length xs)
18   & Map.elems
19   & map (Data.List.foldl' Data.Bits.setBit Data.Bits.zeroBits)
20   where
21     prepend ~[y] ys = y:ys
```

Let's zoom into lines 6–14. Here, we do a fold with a nested fold over the non-fixed cells of the given block to accumulate the mapping from the digits to the indices of the cells they occur in. We use a `Data.Map.Strict` map as the accumulator. If a digit is not present in the map as a key then we add a singleton list containing the corresponding cell index as the value. If the digit is already present in the map then we prepend the cell index to the list of indices for the digit. So we end up "mutating" the map repeatedly.

Of course, it's not actual mutation because the map data structure we are using is immutable. Each change to the map instance creates a new copy with the addition, which we thread through the fold operation, and we get the final copy at the end. This may be the reason of the slowness in this section of the code.

What if, instead of using an immutable data structure for this, we used a mutable one? But how can we do that when we know that Haskell is a pure language? Purity means that all code must be referentially transparent, and mutability certainly isn't. It turns out, there is an escape hatch to mutability in Haskell. Quoting the relevant section from the book Real World Haskell:

> Haskell provides a special monad, named `ST`, which lets us work safely with mutable state. Compared to the `State` monad, it has some powerful added capabilities.
>
> - We can *thaw* an immutable array to give a mutable array; modify the mutable array in place; and freeze a new immutable array when we are done.
> - We have the ability to use *mutable references*. This lets us implement data structures that we can modify after construction, as in an imperative language. This ability is vital for some imperative data structures and algorithms, for which similarly efficient purely functional alternatives have not yet been discovered.

So if we use a mutable map in the `ST` monad, we may be able to get rid of this bottleneck. But, we can actually do better! Since the keys of our map are digits `1`–`9`, we can use a mutable vector to store the indices. In fact, we can go one step even further and store the indices as a BitSet as `Word16` because they also range from 1 to 9, and are unique for a block. This lets us use an unboxed mutable vector. What is *unboxing* you ask? Quoting from the GHC docs:

> Most types in GHC are boxed, which means that values of that type are represented by a pointer to a heap object. The representation of a Haskell `Int`, for example, is a two-word heap object. An unboxed type, however, is represented by the value itself, no pointers or heap allocation are involved.

When combined with vector, unboxing of values means the whole vector is stored as single byte array, avoiding pointer redirections completely. This is more memory efficient and allows better usage of caches[8]. Let's rewrite `exclusivePossibilities` using `ST` and unboxed mutable vectors.

First we write the core of this operation, the function `cellIndicesList` which take a list of cells and returns the digit to cell indices mapping. The mapping is returned as a list. The zeroth value in this list is the indices of the cells which have `1` as a possible digit, and so on. The indices themselves are packed as BitSets. If the bit 1 is set then the first cell has a particular digit. Let's say it returns `[0,688,54,134,0,654,652,526,670]`. In 10-bit binary it is:

```
[0000000000, 1010110000, 0000110110, 0010000110, 0000000000, 1010001110, 1010001100,
1000001110, 1010011110]
```

We can arrange it in a table for further clarity:

Digits	Cell 9	Cell 8	Cell 7	Cell 6	Cell 5	Cell 4	Cell 3	Cell 2	Cell 1
1	0	0	0	0	0	0	0	0	0
2	1	0	1	0	1	1	0	0	0
3	0	0	0	0	1	1	0	1	1
4	0	0	1	0	0	0	0	1	1
5	0	0	0	0	0	0	0	0	0
6	1	0	1	0	0	0	1	1	1
7	1	0	1	0	0	0	1	1	0
8	1	0	0	0	0	0	1	1	1
9	1	0	1	0	0	1	1	1	1

If the value of the intersection of a particular digit and a particular cell index in the table is set to 1, then the digit is a possibility in the cell, else it is not. Here's the code:

```
1  cellIndicesList :: [Cell] -> [Data.Word.Word16]
2  cellIndicesList cells =
3    Data.Vector.Unboxed.toList $ Control.Monad.ST.runST $ do
4      vec <- Data.Vector.Unboxed.Mutable.replicate 9 Data.Bits.zeroBits
5      ref <- Data.STRef.newSTRef (1 :: Int)
6      Control.Monad.forM_ cells $ \cell -> do
7        i <- Data.STRef.readSTRef ref
8        case cell of
9          Fixed _ -> return ()
10         Possible xs -> Control.Monad.forM_ [0..8] $ \d ->
11           Control.Monad.when (Data.Bits.testBit xs (d+1)) $
12             Data.Vector.Unboxed.Mutable.unsafeModify vec (`Data.Bits.setBit` i) d
13       Data.STRef.writeSTRef ref (i+1)
14     Data.Vector.Unboxed.unsafeFreeze vec
```

The whole mutable code runs inside the `runST` function. `runST` take an operation in `ST` monad and executes it, making sure that the mutable references created inside it cannot escape the scope of `runST`. This is done using a type-system trickery called Rank-2 types.

Inside the `ST` operation, we start with creating a mutable vector of `Word16`s of size 9 with all its values initially set to zero. We also initialize a mutable reference to keep track of the cell index we are on. Then we run two nested for loops, going over each cell and each digit `1`–`9`, setting the right bit of the right index of the mutable vector. During this, we mutate the vector directly using the `Data.Vector.Unboxed.Mutable.unsafeModify` function. At the end of the `ST` operation, we freeze the mutable vector to return an immutable version of it. Outside `runST`, we convert the immutable vector to a list. Notice how this code is quite similar to how we'd write it in imperative programming languages like C or Java[9].

It is easy to use this function now to rewrite `exclusivePossibilities`:

```
1    exclusivePossibilities :: [Cell] -> [Data.Word.Word16]
2    exclusivePossibilities row =
3      row
4  -   & zip [1..9]
5  -   & filter (isPossible . snd)
6  -   & Data.List.foldl'
7  -       (\acc ~(i, Possible xs) ->
8  -         Data.List.foldl'
9  -           (\acc' n -> if Data.Bits.testBit xs n
10 -                       then Map.insertWith prepend n [i] acc'
11 -                       else acc')
12 -           acc
13 -           [1..9])
14 -       Map.empty
15 +   & cellIndicesList
16 +   & zip [1..9]
17 -   & Map.filter ((< 4) . length)
18 -   & Map.foldlWithKey' (\acc x is -> Map.insertWith prepend is [x] acc) Map.empty
19 -   & Map.filterWithKey (\is xs -> length is == length xs)
20 +   & filter (\(_, is) -> let p = Data.Bits.popCount is in p > 0 && p < 4)
21 +   & Data.List.foldl' (\acc (x, is) -> Map.insertWith prepend is [x] acc) Map.empty
22 +   & Map.filterWithKey (\is xs -> Data.Bits.popCount is == length xs)
23     & Map.elems
24     & map (Data.List.foldl' Data.Bits.setBit Data.Bits.zeroBits)
25     where
26       prepend ~[y] ys = y:ys
```

We replace the nested two-fold operation with `cellIndicesList`. Then we replace some map related function with the corresponding list ones because `cellIndicesList` returns a list. We also replace the `length` function call on cell indices with `Data.Bits.popCount` function call as the indices are represented as `Word16` now.

That is it. Let's build and run it now:

```
$ stack build
$ cat sudoku17.txt | time stack exec sudoku > /dev/null
      35.04 real        34.84 user         0.24 sys
```

That's a 1.6x speedup over the map-and-fold based version. Let's check what the profiler has to say:

Cost Centre	Src	%time	%alloc
`cellIndicesList.\.\`	Sudoku.hs:(88,11)-(89,81)	10.7	6.0
`primitive`	Control/Monad/Primitive.hs:195:3-16	7.9	6.9
`pruneCells`	Sudoku.hs:(113,1)-(138,53)	7.5	6.4
`cellIndicesList`	Sudoku.hs:(79,1)-(91,40)	7.4	10.1
`basicUnsafeIndexM`	Data/Vector.hs:278:3-62	7.3	0.5
`pruneCells.pruneCell`	Sudoku.hs:(120,5)-(121,83)	6.8	2.0
`exclusivePossibilities`	Sudoku.hs:(94,1)-(104,26)	6.5	9.7
`pruneCells.pruneCell.\`	Sudoku.hs:121:48-83	6.1	2.0
`cellIndicesList.\`	Sudoku.hs:(83,42)-(90,37)	5.5	3.5
`pruneCells.cells`	Sudoku.hs:115:5-40	5.0	10.4

The run time is spread quite evenly over all the functions now and there are no hotspots anymore. We stop optimizating at this point[10]. Let's see how far we have come up.

Comparison of Implementations

Below is a table showing the speedups we got with each new implementation:

Implementation	Run Time (s)	Incremental Speedup	Cumulative Speedup
Simple	47450	1x	1x
Exclusive Pruning	258.97	183.23x	183x
BitSet	69.44	3.73x	683x
Vector	57.67	1.20x	823x
Mutable Vector	35.04	1.65x	1354x

The first improvement over the simple solution got us the most major speedup of 183x. After that, we followed the profiler, fixing bottlenecks by using the right data structures. We got quite significant speedup over the naive list-based solution, leading to drop in the run time from 259 seconds to 35 seconds. In total, we have done more than a thousand times improvement in the run time since the first solution!

Conclusion

In this post, we improved upon our list-based Sudoku solution from the last time. We profiled the code at each step, found the bottlenecks and fixed them by choosing the right data structure for the case. We ended up using BitSets and Vectors — both immutable and mutable varieties — for the different parts of the code. Finally, we sped up our program by 7.4 times. Can we go even faster? How about using all those other CPU cores which have been lying idle? Come back for the next post in this series where we'll explore the parallel programming facilities in Haskell. The code till now is available here. Discuss this post on r/haskell or leave a comment.

Footnotes

1. All the runs were done on my MacBook Pro from 2014 with 2.2 GHz Intel Core i7 CPU and 16 GB memory.↩
2. A lot of the code in this post references the code from the previous posts, including showing diffs. So, please read the previous posts if you have not already done so.↩
3. Notice the British English spelling of the word “Centre”. GHC was originally developed in University of Glasgow in Scotland.↩
4. The code for the BitSet based implementation can be found here.↩
5. This article on School of Haskell goes into details about performance of vectors vs. lists. There are also these benchmarks for sequence data structures in Haskell: lists, vectors, seqs, etc.↩
6. We see Haskell’s laziness at work here. In the code for the `fixM` function, the `(==)` function is nested inside the `(>>=)` function, but because of laziness, they are actually evaluated in the reverse order. The evaluation of parameters for the `(==)` function causes the `(>>=)` function to be evaluated.↩
7. The code for the vector based implementation can be found here.↩
8. Unboxed vectors have some restrictions on the kind of values that can be put into them but `Word16` already follows those restrictions so we are good.↩
9. Haskell can be a pretty good imperative programming language using the `ST` monad. This article shows how to implement some algorithms which require mutable data structures in Haskell.↩
10. The code for the mutable vector based implementation can be found here.↩

