8 Sudoku (fit the second)

The naïve solver from the previous lecture

```
> solve = filter legal . expand . freedoms
```

was too inefficient because it might have to consider as many as $9^{9\times 9-17}$ possible matrices of choices.

8.1 Pruning

One way of cutting down on the scale of the search is to remove from the freedoms in each cell those elements that are already committed to another cell in the same row, column or box. We need a function *prune* which satisfies

$$\mathit{filter}\ \mathit{legal} \cdot \mathit{expand} \quad = \quad \mathit{filter}\ \mathit{legal} \cdot \mathit{expand} \cdot \mathit{prune}$$

But how?

A single row can be pruned by

```
> pruneRow :: Row Freedoms -> Row Freedoms
> pruneRow row = map (remove (ones row)) row
```

removing from each cell of the row any element that is already fixed somewhere in that row. The committed values are the ones that appear as a singleton

```
> ones :: [[a]] -> [a]
> ones row = [ d | [d] <- row ]
```

or, if you prefer (and I might do so)

```
> ones = map head . filter singleton
```

Removing elements from a cell is almost a filter,

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

but had better leave the fixed choices themselves in place.

This function satisfies

$$\mathit{filter}\ \mathit{nodups} \cdot \mathit{cp} = \mathit{filter}\ \mathit{nodups} \cdot \mathit{cp} \cdot \mathit{pruneRow}$$

because any element of the Cartesian product of a row of choices contains every one of the singletons, and so can only be free of duplicates if it does not choose one of these from any other cell.

8.2 Promoting to rows, columns and boxed

Writing n for *nodups* and r, c and b for *rows*, *cols* and *boxs*, recall that

$$legal\ g \quad = \quad all\ n\ (r\ g)\ \&\&\ all\ n\ (c\ g)\ \&\&\ all\ n\ (b\ g)$$

So

$$filter\ legal \quad = \quad filter\ (all\ n \cdot b) \cdot filter\ (all\ n \cdot c) \cdot filter\ (all\ n \cdot r)$$

and if d is one of r, c or b,

$$\begin{aligned}
& filter\ (all\ n \cdot d) \cdot expand \\
= \quad & \{\ map\ id = id \text{ and } d \cdot d = id\ \} \\
& \underline{map\ (d \cdot d)} \cdot filter\ (all\ n \cdot d) \cdot expand \\
= \quad & \{\ map\ f \cdot map\ g = map\ (f \cdot g)\ \} \\
& map\ d \cdot \underline{map\ d \cdot filter\ (all\ n \cdot d)} \cdot expand \\
= \quad & \{\ map\ f \cdot filter\ (p \cdot f) = filter\ p \cdot map\ f\ \} \\
& map\ d \cdot filter\ (all\ n) \cdot \underline{map\ d \cdot expand} \\
= \quad & \{\ map\ d \cdot expand = expand \cdot d\ \} \\
& map\ d \cdot filter\ (all\ n) \cdot \underline{expand} \cdot d \\
= \quad & \{\ \text{definition of } expand = cp \cdot map\ cp\ \} \\
& map\ d \cdot \underline{filter\ (all\ n) \cdot cp} \cdot map\ cp \cdot d \\
= \quad & \{\ filter\ (all\ p) \cdot cp = cp \cdot map\ (filter\ p)\ \} \\
& map\ d \cdot cp \cdot \underline{map\ (filter\ n) \cdot map\ cp} \cdot d \\
= \quad & \{\ map\ f \cdot map\ g = map\ (f \cdot g)\ \} \\
& map\ d \cdot cp \cdot map\ (\underline{filter\ n \cdot cp}) \cdot d \\
= \quad & \{\ filter\ n \cdot cp = filter\ n \cdot cp \cdot pruneRow\ \} \\
& map\ d \cdot \underline{cp \cdot map\ (filter\ n \cdot cp \cdot pruneRow)} \cdot d \\
= \quad & \{\ \text{reversing four steps}\ \} \\
& map\ d \cdot filter\ (all\ n) \cdot expand \cdot map\ pruneRow \cdot d \\
= \quad & \{\ map\ d \cdot map\ d \text{ is the identity}\ \} \\
& map\ d \cdot \underline{filter\ (all\ n) \cdot map\ d} \cdot map\ d \cdot expand \cdot map\ pruneRow \cdot d \\
= \quad & \{\ filter\ p \cdot map\ f = map\ f \cdot filter\ (p \cdot f)\ \} \\
& \underline{map\ d \cdot map\ d} \cdot filter\ (all\ n \cdot d) \cdot \underline{map\ d \cdot expand} \cdot map\ pruneRow \cdot d \\
= \quad & \{\ map\ d \cdot map\ d \text{ is the identity, and } map\ d \cdot expand = expand \cdot d\ \} \\
& filter\ (all\ n \cdot d) \cdot expand \cdot d \cdot map\ pruneRow \cdot d
\end{aligned}$$

Because the three filters commute with each other we can use this calculation three times to deduce that

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

will do.

This suggests that we would be better to try

```
> prunesolve :: Solver
> prunesolve = filter legal . expand . prune . freedoms
```

Sadly this doesn't help much.

However, as in other walks of life, one *prune* has only a small effect. The same calculation shows that we can have two *prune*s, three *prune*s, ... as many *prune*s as we need. What if we have as many as are needed for one more to make no differnce?

```
> bowlsolve :: Solver
> bowlsolve =
>     filter legal . expand . repeatedly prune . freedoms
```

This turns out to be enough to solve easy problems.

The function *repeatedly* is not standard: it might have been defined by

```
repeatedly f x | fx == x   = x
               | otherwise = repeatedly f fx
                 where fx = f x
```

but in the lecture script I have

```
> repeatedly :: Eq a => (a -> a) -> a -> a
> repeatedly f =
>     fst . head . dropWhile (uncurry (/=)) . pairs . iterate f
>   where pairs xs = xs `zip` tail xs
```

We can be sure that *repeatedly prune* cannot go on for ever looking for a fixed point. (Why?)

Unfortunately, however many times we prune, the more devious Sudoku problems will still have so much freedom left that this solver takes too long.

8.3 Parsimonious expansion

Another strategy that human solvers apply is to look for cells that are highly constrained, and consider the consequences of making each possible choice there.

Suppose we can define

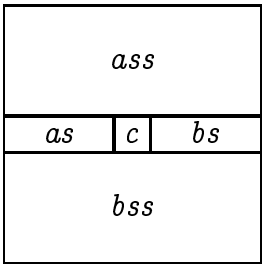
```
expand1 :: Matrix Freedoms -> [ Matrix Freedoms ]
```

to expand only one cell. This ought to satisfy

$$expand \quad = \quad concat \cdot map\ expand \cdot expand1$$

at least up to permutation of the answer.

A good choice of cell on which to perform this expansion is any with a minimal number of choices, but not yet fixed. There might be no choices, in which case there is no expansion; or there might be at least two.




```
> expand1 :: Matrix Freedoms -> [ Matrix Freedoms ]
> expand1 fss = [ ass++[as++[[c]]++bs]++bss | c <- cell ]
>    where
>      (ass,row:bss) = break (any optimal) fss
>      (as, cell:bs) = break optimal row
>      optimal cell  = length cell == n
>      n             = minimum (lengths fss)
> lengths = filter (/=1) . map length . concat
```

where the predefined function *break* satisfies

$$break\ p\ xs \quad = \quad (takeWhile\ (not \cdot p)\ xs, dropWhile\ (not \cdot p)\ xs)$$

and divides a list into two just before the first p-satisfying element of the list.

Some care is needed: if there are singletons in all of the cells of the matrix, this function will be undefined. So

$$expand \quad = \quad concat \cdot map\ expand \cdot expand1$$

can only hold if there is at least one non-singleton cell (possibly an empty one).

Say that a matrix of freedoms is *complete* if all cells are singletons, and *safe* if the matrix of singletons is free of duplicates. Unsafe matrices cannot lead to a solution. A complete and safe matrix corresponds to a unique legal grid.

```
> complete :: Matrix Freedoms -> Bool
> complete = all (all singleton)

> safe :: Matrix Freedoms -> Bool
> safe fss = all (nodups.ones) (rows fss) &&
>            all (nodups.ones) (cols fss) &&
>            all (nodups.ones) (boxs fss)
```

Assuming that it is applied to a matrix that is safe but incomplete,

$$
\begin{aligned}
& \mathit{filter}\ \mathit{legal} \cdot \mathit{expand} \\
= \quad & \{ \text{construction of } \mathit{expand1} \} \\
& \mathit{filter}\ \mathit{legal} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{expand} \cdot \mathit{expand1} \\
= \quad & \{ \mathit{filter}\ p \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{filter}\ p) \} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{filter}\ \mathit{legal}) \cdot \mathit{map}\ \mathit{expand} \cdot \mathit{expand1} \\
= \quad & \{ \mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f \cdot g) \} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{filter}\ \mathit{legal} \cdot \mathit{expand}) \cdot \mathit{expand1} \\
= \quad & \{ \text{construction of } \mathit{prune} \} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{filter}\ \mathit{legal} \cdot \mathit{expand} \cdot \mathit{prune}) \cdot \mathit{expand1} \\
= \quad & \{ \text{let } \mathit{search} = \mathit{filter}\ \mathit{legal} \cdot \mathit{expand} \cdot \mathit{prune} \} \\
& \mathit{concat} \cdot \mathit{map}\ \mathit{search} \cdot \mathit{expand1}
\end{aligned}
$$

This suggests we can write a solver

```
> solve :: Solver
> solve = search . freedoms
```

where *search* uses this derivation to solve an incomplete problem

```
search fss | not (complete pss) = (concat . map search . expand1) pss
           | complete pss       = [ map (map head) pss | safe pss ]
             where pss = prune fss
```

and in the case of a completed puzzle, either to return it or not according to whether or not it is safe (and so legal).

Since, once the committed squares become unsafe, there is no way of filling in the other squares in a safe way, it would be more efficient to abandon the search as soon as the problem becomes unsafe

```
> search :: Matrix Freedoms -> [ Grid ]
> search fss | not (safe pss) = []
>            | complete pss   = [ map (map head) pss ]
>            | otherwise      = (concat . map search . expand1) pss
>              where pss = prune fss
```

and this solves problems much faster than I can.

Exercises

8.1 Aligning text in columns involves *justification*: perhaps to the right or left. One way of doing involves padding strings to a given length:

```
*Main> rjustify 10 "word"
"      word"
*Main> ljustify 10 "word"
"word      "
```

Define functions

```
> rjustify :: Int -> String -> String
> ljustify :: Int -> String -> String
```

to do this. What do your functions do if the string is wider than the target length? Is that what you would want, and if not how would you do it differently?

8.2 Suppose we represent an $n \times m$ matrix by a list of n rows, each of which is a list of m elements. These matrices will be elements of

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

that are, additionally to what the type says, rectangular and non-empty. Without writing any new recursive definitions:

1. define $scale :: Num\ a \Rightarrow a \rightarrow Matrix\ a \rightarrow Matrix\ a$ which multiplies each element of a matrix by a scalar (the qualification $Num\ a$ in the type means that *scale* can use arithmetic on values of type a);
2. define a function $dot :: Num\ a \Rightarrow [a] \rightarrow [a] \rightarrow a$ which calculates the dot-product of two vectors of the same length;
3. define $add :: Num\ a \Rightarrow Matrix\ a \rightarrow Matrix\ a \rightarrow Matrix\ a$ which adds to matrices (of the same size as each other);
4. define $mul :: Num\ a \Rightarrow Matrix\ a \rightarrow Matrix\ a \rightarrow Matrix\ a$ which multiplies matrices of appropriately matching sizes;
5. define $table :: Show\ a \Rightarrow Matrix\ a \rightarrow String$ that translates a matrix (of printable elements) into a string that can be printed to show the matrix with each element right-justified in a column just wide enough to contain each of its elements. You may want to use the predefined *unwords* and *unlines*.

```
*Main> putStr (table [[1,-500,-4], [100,15043,6], [5,3,10]])
  1  -500 -4
100 15043  6
  5     3 10
```