

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

$$\text{filter legal} \cdot \text{expand} = \text{filter legal} \cdot \text{expand} \cdot \text{prune}$$

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

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

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 *boxes*, recall that

$$\text{legal } g = \text{all } n \ (r \ g) \ \&\& \ \text{all } n \ (c \ g) \ \&\& \ \text{all } n \ (b \ g)$$

So

$$\text{filter } \text{legal} = \text{filter } (\text{all } n \cdot b) \cdot \text{filter } (\text{all } n \cdot c) \cdot \text{filter } (\text{all } n \cdot r)$$

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

$$\begin{aligned} & \text{filter } (\text{all } n \cdot d) \cdot \text{expand} \\ = & \{ d \cdot d = \text{id} \text{ and } \text{map } \text{id} = \text{id} \} \\ & \text{map } (d \cdot d) \cdot \text{filter } (\text{all } n \cdot d) \cdot \text{expand} \\ = & \{ \text{map } f \cdot \text{map } g = \text{map } (f \cdot g) \} \\ & \text{map } d \cdot \text{map } d \cdot \text{filter } (\text{all } n \cdot d) \cdot \text{expand} \\ = & \{ \text{map } f \cdot \text{filter } (p \cdot f) = \text{filter } p \cdot \text{map } f \} \\ & \text{map } d \cdot \text{filter } (\text{all } n) \cdot \text{map } d \cdot \text{expand} \\ = & \{ \text{map } d \cdot \text{expand} = \text{expand} \cdot d \} \\ & \text{map } d \cdot \text{filter } (\text{all } n) \cdot \text{expand} \cdot d \\ = & \{ \text{definition of } \text{expand} = \text{cp} \cdot \text{map } \text{cp} \} \\ & \text{map } d \cdot \text{filter } (\text{all } n) \cdot \text{cp} \cdot \text{map } \text{cp} \cdot d \\ = & \{ \text{filter } (\text{all } p) \cdot \text{cp} = \text{cp} \cdot \text{map } (\text{filter } p) \} \\ & \text{map } d \cdot \text{cp} \cdot \text{map } (\text{filter } n) \cdot \text{map } \text{cp} \cdot d \\ = & \{ \text{map } f \cdot \text{map } g = \text{map } (f \cdot g) \} \\ & \text{map } d \cdot \text{cp} \cdot \text{map } (\text{filter } n \cdot \text{cp}) \cdot d \\ = & \{ \text{filter } n \cdot \text{cp} = \text{filter } n \cdot \text{cp} \cdot \text{pruneRow} \} \\ & \text{map } d \cdot \text{cp} \cdot \text{map } (\text{filter } n \cdot \text{cp} \cdot \text{pruneRow}) \cdot d \\ = & \{ \text{reversing four steps} \} \\ & \text{map } d \cdot \text{filter } (\text{all } n) \cdot \text{expand} \cdot \text{map } \text{pruneRow} \cdot d \\ = & \{ \text{map } d \cdot \text{map } d \text{ is the identity} \} \\ & \text{map } d \cdot \text{filter } (\text{all } n) \cdot \text{map } d \cdot \text{map } d \cdot \text{expand} \cdot \text{map } \text{pruneRow} \cdot d \\ = & \{ \text{filter } p \cdot \text{map } f = \text{map } f \cdot \text{filter } (p \cdot f) \} \\ & \text{map } d \cdot \text{map } d \cdot \text{filter } (\text{all } n \cdot d) \cdot \text{map } d \cdot \text{expand} \cdot \text{map } \text{pruneRow} \cdot d \\ = & \{ \text{map } d \cdot \text{map } d \text{ is the identity, and } \text{map } d \cdot \text{expand} = \text{expand} \cdot d \} \\ & \text{filter } (\text{all } n \cdot d) \cdot \text{expand} \cdot d \cdot \text{map } \text{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 boxes . pruneBy cols . pruneBy rows
>       where pruneBy f = f . map pruneRow . f
```

will do.

This suggests that we would be better to try

```
> prunesolver :: Solver
> prunesolver = 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 *prunes*, three *prunes*, ... as many *prunes* as we need. What if we have as many as are needed for one more to make no difference?

```
> 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 = concat . map expand . expand1
```

at least, up to permutation of the answer.

<i>ass</i>		
<i>as</i>	<i>c</i>	<i>bs</i>
<i>bss</i>		

A good choice of cell on which to perform this expansion is one with a minimal number of choices, but not yet fixed. There might be zero 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*

```
break :: (a -> Bool) -> [a] -> ([a], [a])
break p xs = (takeWhile (not.p) xs, dropWhile (not.p) xs)
```

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 = concat . map expand . 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,

```

    filter legal . expand
= { construction of expand1 }
    filter legal . concat . map expand . expand1
= { filter p . concat = concat . map (filter p) }
    concat . map (filter legal) . map expand . expand1
= { map f . map g = map (f . g) }
    concat . map (filter legal . expand) . expand1
= { construction of prune }
    concat . map (filter legal . expand . prune) . expand1
= { let search = filter legal . expand . prune }
    concat . map search . expand1

```

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.