A Sudoku Solver – Pruning (3A)

- Richard Bird Implementation

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

Thinking Functionally with Haskell, R. Bird

https://wiki.haskell.org/Sudoku

http://cdsoft.fr/haskell/sudoku.html

https://gist.github.com/wvandyk/3638996

http://www.cse.chalmers.se/edu/year/2015/course/TDA555/lab3.html

Single-Cell Expansion

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

> pruneRow :: Row Choices -> Row Choices
> pruneRow row = map (remove ones) row
>  where ones = [d | [d] <- row]
```

Single-Cell Expansion

```
solve :: Grid -> [Grid]
solve = filter valid . expand. Choices

prune :: Matrix [Digit] -> Matrix [Digit]
filter valid . Expand = filter valid . Expand

pruneRow :: Row [Digit] -> Row [Digit]
pruneRow row = map (remove fixed) row
      where fixed = [d | [d] ← row]

remove :: [Digit] -> [Digit] -> [Digit]
remove ds [x] = [x]
remove ds xs = filter (`notElem` ds) xs

notElem :; (Eq a_ => a -> [a] -> Bool
notElem x xs = all (/= x) xs
```

Single-Cell Expansion

```
pruneRow [[6], [1,2], [3], [1,3,4], [5,6]]
[[6], [1,2], [3], [1,4], [5]]

PruneRow [[6], [3,6], [3], [1,3,4], [4]]
[[6], [], [3], [1], [4]]

filter nodups . cp = filter nodups . cp . PruneRow

filter (p. f) = map f . filter p . map f
filter (p. f) map f = map f . filter p
```

Single-Cell Expansion

```
map f. filter p . map f
= map f . filter (p . f)

map f . map f . filter (p . f)
= filter (p . f)

filter valid . expand
= filter (all nodups . boxs) .
   filter (all nodups . cols) .
   filter (all nodups . rows) . expand
```

Single-Cell Expansion

```
filter (all nodups  . boxs) . expand
= map boxs . filter (all nodups) . map boxs . expand
= map boxs . filter (all nodups) . cp . map cp . boxs
= map boxs . cp . map (filter nodups) .map cp . boxs
= map boxs .cp . map (filter nodups . cp) . boxs

boxs . boxs = id
map boxs . expand = expand . boxs
filter (all p) . cp = cp . map . (filter p)

filter nodups . cp = filter nodups . cp . prunerow

map boxs . cp . map (filter nodups . cp . prunerow) . boxs
```

Single-Cell Expansion

```
map boxs . cp . map (filter nodups . cp . prunerow) . box =
map boxs .cp . map (filter nodups) . map (cp . prunerow) . boxs =
map boxs . filter (all nodups) . cp . map (cp . prunerow) . boxs =
map boxs . filter (all nodups) . cp . map cp . map prunerow . boxs =
map boxs. filter (all nodups) . expand . map prunerow . boxs =
filter (all nodups . boxs) . map boxs . expand . map prunerow . boxs =
filter (all nodups . boxs) . expand . bosx . map prunerow . boxs =
filter (all nodups . boxs) .expand . pruneby boxs =

filter (all nodups . boxs) . expand =
filter (all nodups . boxs) . expand . pruneby boxs

filter valid . expand = filter valid . expand . prune

prune = prunby boxs .pruneby cols . pruneby rows
```

Single-Cell Expansion

```
solve = filter valid . expand . prune . choices

many :: (eq a) => (a -> a) -> a -> a
many f x = if x == y then x else many f y
     where y = f x

solve = filter valid . expand . many prune . choices
```

Single-Cell Expansion

```
expand1   :: Matrix Choices -> [Matrix Choices]
expand1 rows =
  [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
  where
  (rows1,row:rows2) = break (any smallest) rows
  (row1,cs:row2)       = break smallest row
  smallest cs             = length cs == n
  n                            = minimum (counts rows)

 counts                      = filter (/=1) . map length . concat
```

Single-Cell Expansion

```
> solve2 :: Grid -> [Grid]
> solve2 =  search . choices

> search :: Matrix Choices -> [Grid]
> search cm
>  |not (safe pm)  = []
>  |complete pm    = [map (map head) pm]
>  |otherwise      = (concat . map search . expand1) pm
>  where pm = prune cm

> complete :: Matrix Choices -> Bool
> complete = all (all single)

> single [_] = True
> single _   = False
```

Single-Cell Expansion

```
> solve2 :: Grid -> [Grid]
> solve2 =  search . choices

> search :: Matrix Choices -> [Grid]
> search cm
>  |not (safe pm)  = []
>  |complete pm    = [map (map head) pm]
>  |otherwise      = (concat . map search . expand1) pm
>  where pm = prune cm

> complete :: Matrix Choices -> Bool
> complete = all (all single)

> single [_] = True
> single _   = False
```

Single-Cell Expansion

```
> safe :: Matrix Choices -> Bool
> safe cm = all ok (rows cm) &&
>           all ok (cols cm) &&
>           all ok (boxs cm)

> ok row = nodups [d | [d] <- row]
```

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf