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

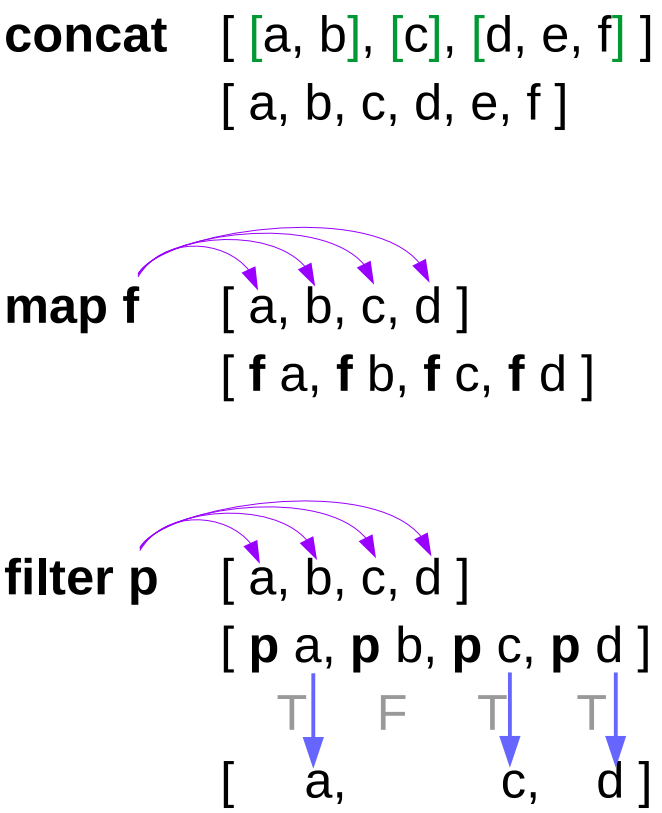
http://www.cse.chalmers.se/edu/year/2015/course/TDA555/lab3.html

concat, map, filter

```
concat :: [[a]] -> [a]
concat []          = []
concat (xs:xss)    = xs ++ concat xss
```

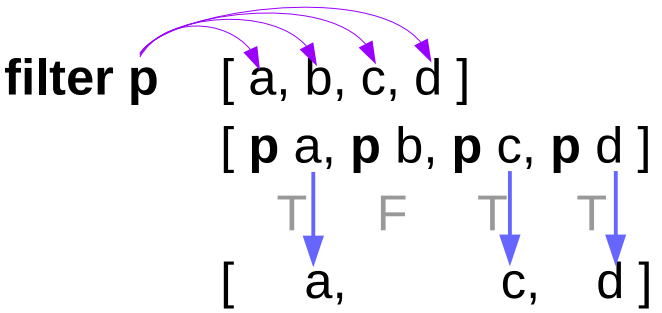
```
map :: (a -> b) -> [a] -> [b]
map f []           = []
map f (xs:xss)     = f xs : map f xss
```

```
filter :: (a ->bool) -> [a] -> [a]
filter p []        = []
filter p (xs:xss)  = if p xs then xs : filter p xss
                     else filter p xss
```

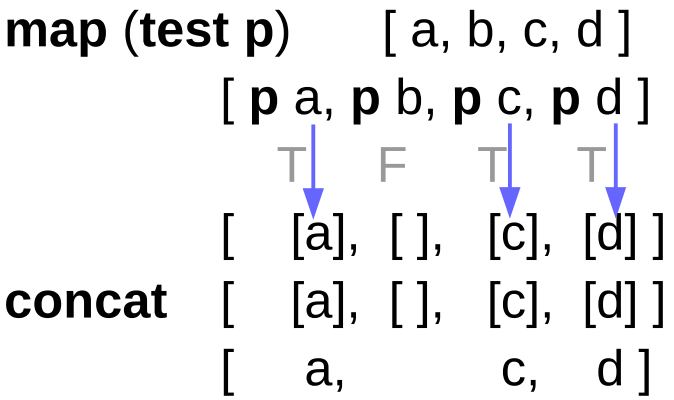


Definitions of filter

```
filter :: (a ->bool) -> [a] -> [a]
filter p []          = []
filter p (xs:xss)    = if p xs then xs : filter p xss
                       else filter p xss
```




```
filter p = concat . map (test p)
test p x = if p x then [x] else [ ]
```



Definitions of filter

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

test p x = if p x then [x] else [ ]
```

```
test p . f x
= test p (f x)
= if p (f x) then [f x] else [ ]
```

```
test (p . f) x = if (p . f) x then [x] else [ ]
= if p (f x) then [x] else [ ]
```

```
map f . test (p . f) x =
= if p (f x) then map f [x] else map f [ ]
= if p (f x) then [f x] else [ ]
```

concat, map, filter

map id = **id**

map (f . g) = **map f . map g**

map id [a, b, c, d]
[**id** a, **id** b, **id** c, **id** d]
[a, b, c, d]

map(f.g) [a, b, c, d]
[**f.g** a, **f.g** b, **f.g** c, **f.g** d]
[**f** (**g** a), **f** (**g** b), **f** (**g** c), **f** (**g** d)]

map g [a, b, c, d]
[**g** a, **g** b, **g** c, **g** d]

map f . map g [a, b, c, d]
map f [**g** a, **g** b, **g** c, **g** d]
[**f** (**g** a), **f** (**g** b), **f** (**g** c), **f** (**g** d)]

concat, map, filter

f . head = head . map f

f . head [a, b, c, d] = **f** a
head . map f [a, b, c, d] =
head [**f** a, **f** b, **f** c, **f** d] = **f** a

map f . tail = tail . map f

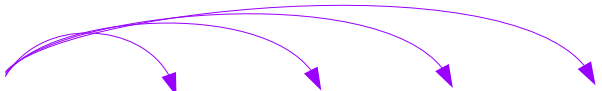
map f . tail [a , b, c, d] =
map f [b, c, d] = [**f** b, **f** c, **f** d]
tail . **map f** [a, b, c, d] =
tail [**f** a, **f** b, **f** c, **f** d] = [**f** b, **f** c, **f** d]

map f . concat =
concat . map (map f)

map f . concat [[a], [b], [c], [d]] =
map f . [a , b, c, d] = [**f** a, **f** b, **f** c, **f** d]
concat . **map** (**map f**) [[a], [b], [c], [d]] =
concat . [**map f** [a], **map f** [b], **map f**[c], **map f**[d]] =
concat [[**f** a], [**f** b], [**f** c], [**f** d]] =[**f** a, **f** b, **f** c, **f** d]

concat . concat

concat . map concat = concat . concat



concat . **map concat** [[[a]], [[b]], [[c]], [[d]]] remove inside [ ] first
concat . [**concat** [[a]], **concat** [[b]], **concat** [[c]], **concat** [[d]]]
concat [[a], [b], [c], [d]]
[a, b, c, d]

concat . **concat** [[[a]], [[b]], [[c]], [[d]]] remove outside [ ] first
concat [[a], [b], [c], [d]]
[a, b, c, d]

map f . concat

concat . **map** (**map f**) = **map f** . **concat**

concat . **map** (**map f**) [[a], [b], [c], [d]]
concat [**map f** [a], **map f** [b], **map f** [c], **map f** [d]]
concat [[**f** a], [**f** b], [**f** c], [**f** d]]
[**f** a, **f** b, **f** c, **f** d]

map f . **concat** [[a], [b], [c], [d]]

map f [a, b, c, d]
[**f** a, **f** b, **f** c, **f** d]

Strict Function

f . **head** = **head** . **map f**

f (**head** [ ]) = **head** (**map f** [ ]) = **head** [ ] (undefined)

concat, map, filter

```
tail     :: [a] -> [a]
reverse  :: [a] -> [a]
```

```
map f . tail     =     tail . map f
map f . reverse  =  reverse . map f
```

```
head     ::  [a]  ->  a
concat   :: [[a]] -> [a]
```

```
    f . head     =     head . map f
map f . concat   =  concat . map (map f)
concat . concat  =  concat . map concat
```

filter p . map f

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

filter p . map f [a, b, c, d]

filter p . [**f** a, **f** b, **f** c, **f** d]

[**p** (**f** a), **p** (**f** b), **p** (**f** c), **p** (**f** d)]

T F T T

[**f** a , **f** c, **f** d]

map f . filter (**p** . **f**) [a, b, c, d]

[(**p**.**f**) a, (**p**.**f**) b, (**p**.**f**) c, (**p**.**f**) d]

T F T T

map f . [a, c, d]

[**f** a , **f** c, **f** d]

filter p . map f – proof

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

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

= concat . map (test p . f)
= concat . map (map f . test (p . f))

= concat . map (map f) . map (test (p . f))

= map f . concat .  map (test (p . f))

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

filter p = concat . map (test p)
test p x = if p x then [x] else [ ]

map m . map n = map m . n

test p . f = map f . test (p . f)

map m . map n = map m . n

concat . map (map f) = map f . concat

filter p = concat . map (test p)

filter (p.f) with a self-inverse f

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

map f . filter p . map f [a, b, c, d]

map f . filter p [**f** a, **f** b, **f** c, **f** d]

map f . [**p** (**f** a), **p** (**f** b), **p** (**f** c), **p** (**f** d)]

map f [**f** a, **f** c, **f** d]

[**f** (**f** a), **f** (**f** c), **f** (**f** d)]

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

map f . filter p [a, b, c, d]

map f [**p** a, **p** b, **p** c, **p** d]

map f [a, c, d]

[**f** a, **f** c, **f** d]

filter (p.f) with a self-inverse f – proof

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

Assume **f . f = id** ➡ **map f . map f = map (f . f) = map id = id**

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

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

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

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

map and filter

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

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

Assume f . f = id

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

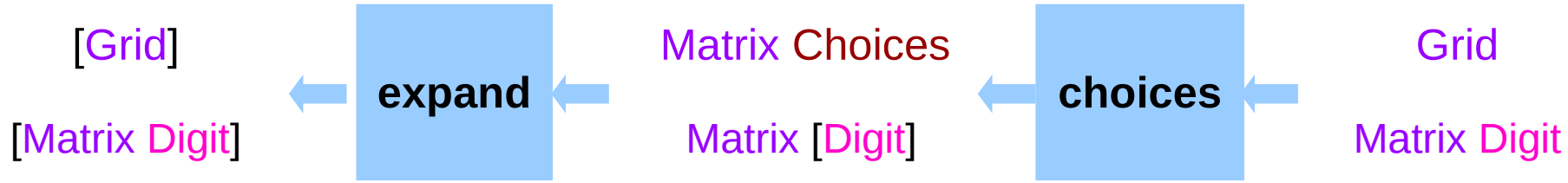
```
filter (p . f) . map f = map f . filter p
```

Solve with pruning

solve :: Grid -> [Grid]
solve = **filter valid** . **expand**. **choices**

filter valid . **expand** = **filter valid** . **expand** . **prune**

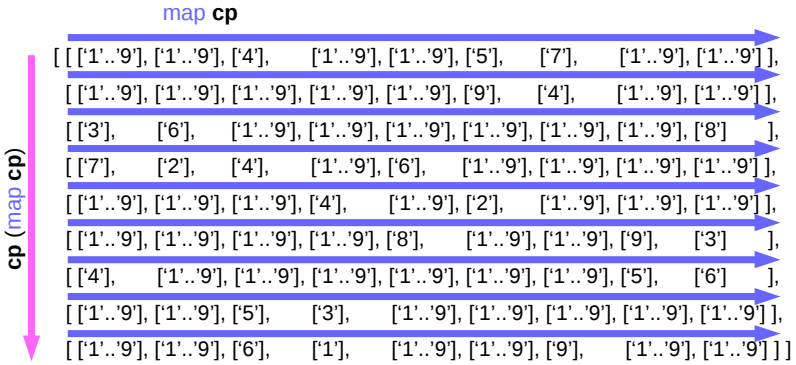
prune :: Matrix [Digit] -> Matrix [Digit]



expand – map cp

Matrix Choices

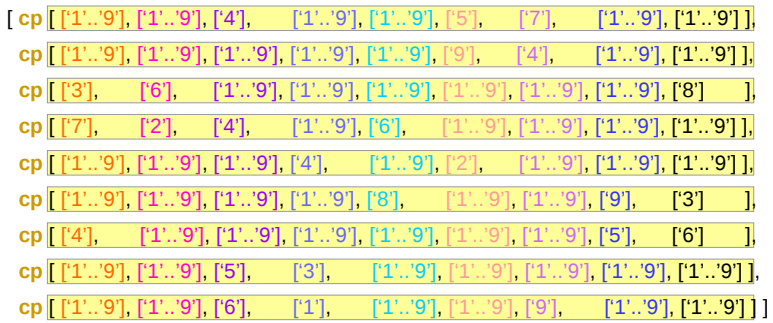
Matrix [Digit]

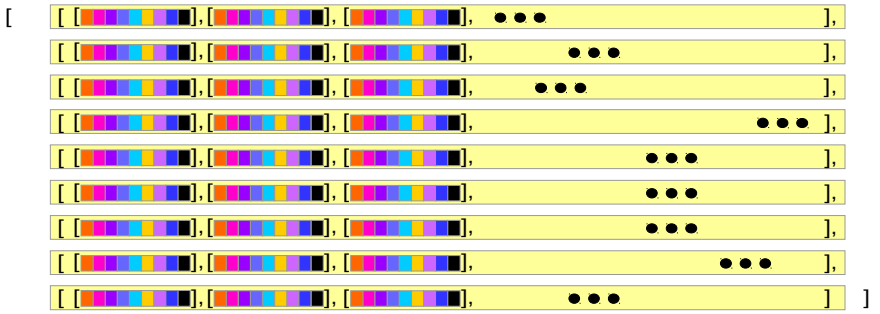


expand :: Matrix Choices -> [Grid]

expand = **cp** . map **cp**

cp . map **cp** = [[[a]]] -> [[[a]]]

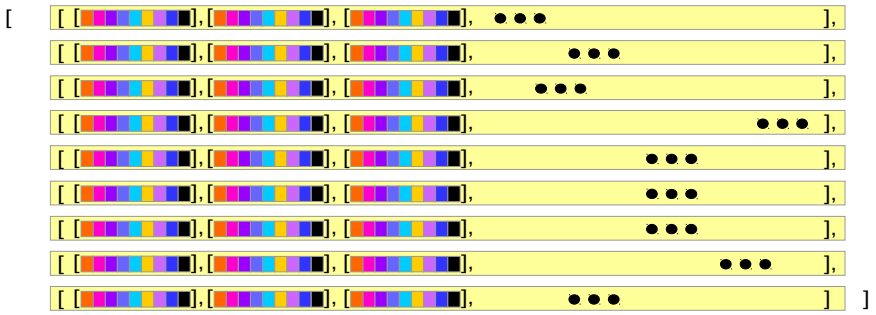




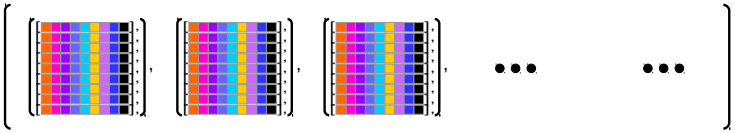
map **cp**

expand – cp . map cp

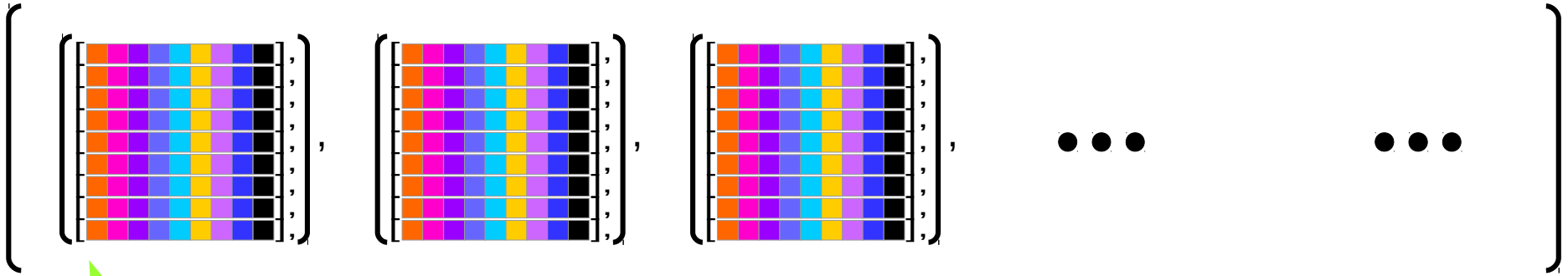
map **cp**



cp . map **cp**



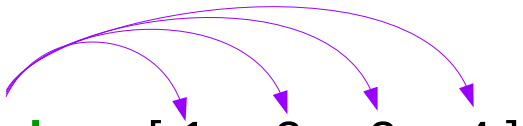
[Grid]
[Matrix Digit]



Grid = Matrix Digit
➡ [Row Digit]
➡ [[Digit]]

Valid

```
valid  :: Grid -> Bool
valid g = all nodups (rows g) &&
          all nodups (cols  g) &&
          all nodups (boxs g)
```

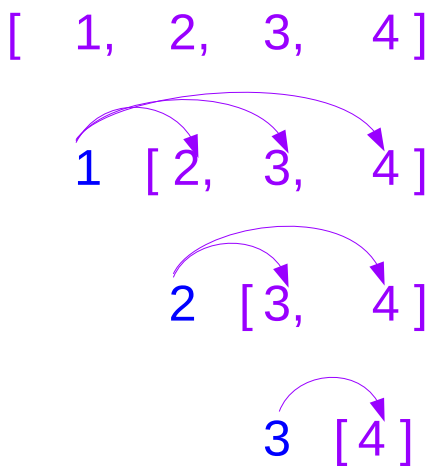


```
all nodups [r1   r2   r3   r4 ]
```

```
(nodups r1) && (nodups r2) && (nodups r3) && (nodups r4)
```

nodups

```
valid  :: Grid -> Bool
valid g = all nodups (rows g) &&
          all nodups (cols  g) &&
          all nodups (boxs g)

nodups :: Eq a => [a] -> Bool
nodups []     = True
nodups (x:xs) = x `notElem` xs && nodups xs
```



notElem

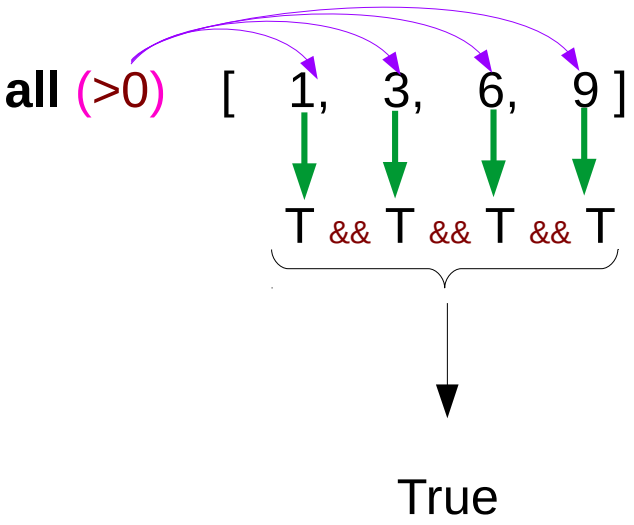
```
nodups :: Eq a => [a] -> Bool
nodups []       = True
nodups (x:xs) = x `notElem` xs && nodups xs

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

```
all (/= 2) [  1,   3,   6,   9 ]
           T && T && T && T     →   True
```

all

all :: (a -> Bool) -> [a] -> Bool

Determines whether all elements of the structure satisfy the predicate.



Infix and Prefix

Using infix functions with **prefix notation** – parenthesis **( )**

(+) 1 2
(*) 3 4

Using prefix functions with **infix notation** – backtick ` `

foldl (+) 0 [1..5]

((+) `**foldl**` 0) [1..5]

https://wiki.haskell.org/Infix_operator

pruneRow

```
pruneRow :: Row [Digit] -> Row [Digit]
pruneRow row = map (remove fixed) row
    where fixed = [d | [d] <- row]          -- single element list – the only choice
```

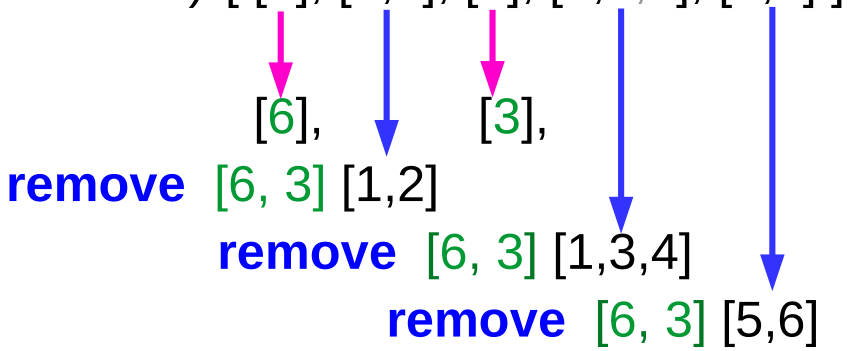
```
pruneRow [ [6], [1,2], [3], [1,3,4], [5,6] ]      -- Fixed →  [6, 3]
[[6], [1,2], [3], [1,4], [5]]

pruneRow [ [6], [3,6], [3], [1,3,4], [4] ]        -- Fixed → [6, 3, 4]
[[6], [], [3], [1], [4]]
```

filter nodups . cp = filter nodups . cp . PruneRow

Remove

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

fixed = [6, 3]
```

map (**remove** fixed) [[6], [1,2], [3], [1,3,4], [5,6]]



Remove

Remove [6, 3] [1,2]

filter (`notElem` [6, 3]) [1,2] → [1,2]

remove [6, 3] [1,3,4]

filter (`notElem` [6, 3]) [1,3,4] → [1,4]

remove [6, 3] [5,6]

filter (`notElem` [6, 3]) [1,3,4] → [1,4]

Remove

```
pruneRow :: Row [Digit] -> Row [Digit]
pruneRow row = map (remove fixed) row
      where fixed = [d | [d] <- row]                -- single element list (the only choice)

fixed = [6, 3]

map (remove fixed) [ [6], [1,2], [3], [1,3,4], [5,6] ]

remove :: [Digit] -> [Digit] -> [Digit]
remove ds [x] = [x]                                 -- do not remove fixed choices
remove ds xs = filter (`notElem` ds) xs


filter (`notElem` ds)       [ [6], [1,2], [3], [1,3,4], [5,6] ]
```

prune

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

f . f = id

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

-- single element list
-- (the only choice) fixed = ones

pruneBy

```
boxes . boxs    = id
cols  . cols    = id
rows  . rows    = id


pruneBy boxs  = boxs  . map pruneRow . boxs
pruneBy cols  = cols  . map pruneRow . cols
pruneBy rows  = rows  . map pruneRow . Rows

pruneBy f = f . map pruneRow . f

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

Filter valid . expand

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

```
valid  :: Grid -> Bool
valid g = all nodups (rows g) &&
          all nodups (cols  g) &&
          all nodups (boxs g)
```

```
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
```

Filter valid . expand

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

```
valid  :: Grid -> Bool
valid g = all nodups (rows g) &&
          all nodups (cols  g) &&
          all nodups (boxs g)
```

```
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
```

map and filter

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

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

Assume f . f = id

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

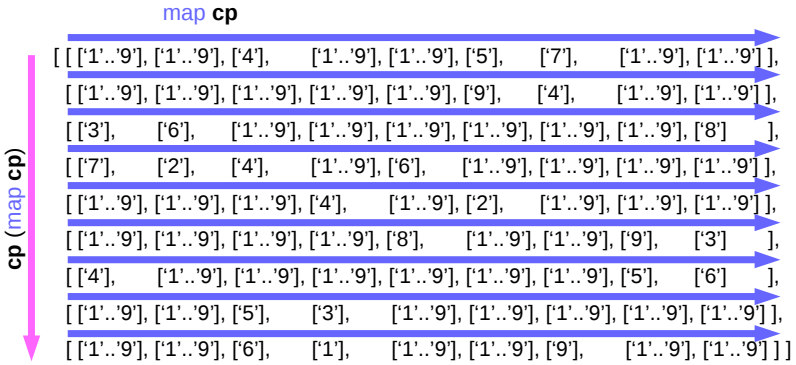
filter (p . f) . map f = map f . filter p

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

expand – map cp

Matrix Choices

Matrix [Digit]



expand :: Matrix Choices -> [Grid]

expand = **cp** . map **cp**

cp . map **cp** = [[[a]]] -> [[[a]]]

map rows . expand = expand . rows
map cols . expand = expand . cols
map boxs . expand = expand . boxs

map boxs . expand
= map boxs . cp . map cp
= cp . map cp . boxs

expand – map cp

```
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
```

```
map boxs . expand
= map boxs . cp . map cp
= cp . map cp . boxs
```

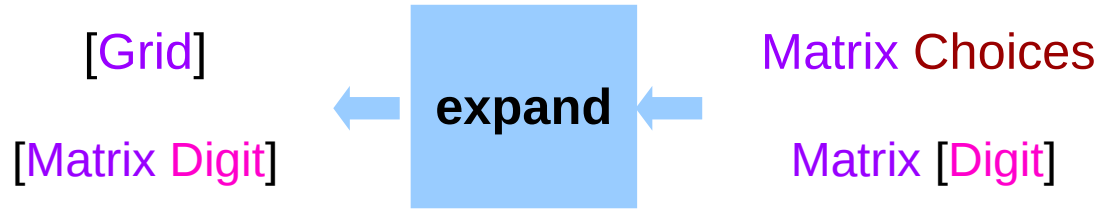
map and filter

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

```
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
```

Expand Laws

```
expand :: Matrix Choices -> [Grid]
expand = cp . map cp

cp . map cp = [ [[a]] ] -> [ [[a]] ]
```



filter (p.f) with a self-inverse f – proof

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

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

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

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

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

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

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

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

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

f . f = id assumed

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

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

map f. filter p . map 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 = pruneBy boxs . pruneBy cols . pruneBy rows

Single-Cell Expansion

```
cp . map (filter p) = filter (all p) . cp
boxs . boxs = id
boxs . expand = expand . boxs
boxs . boxs = id
pruneBy f = f . pruneRow . f
```

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