# A Sudoku Solver – Pruning (3A)

- Richard Bird Implementation

Copyright (c) 2016 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

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

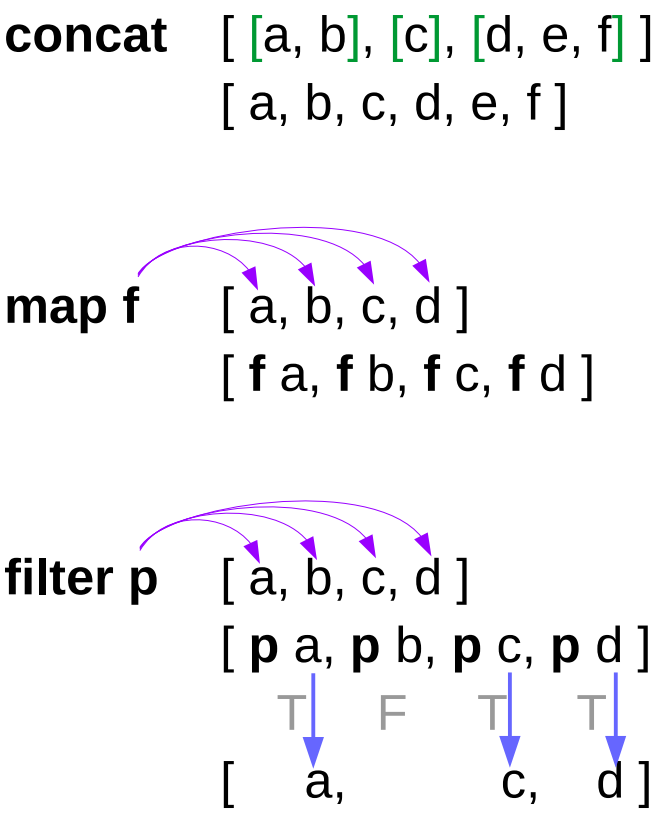
http://www.cse.chalmers.se/edu/year/2015/course/TDA555/lab3.html

# concat, map, filter

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

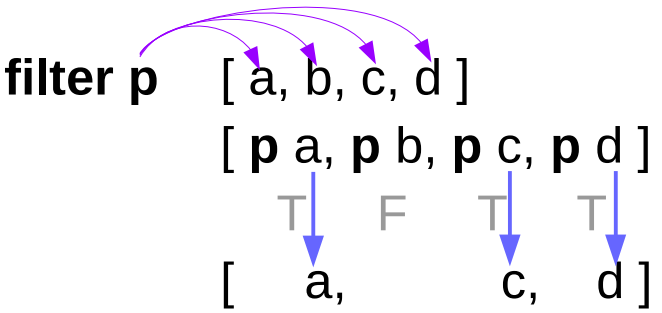
```
map :: (a -> b) -> [a] -> [b]
map f []           = []
map f (xs:xss)     = f xs : map f xss
```

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

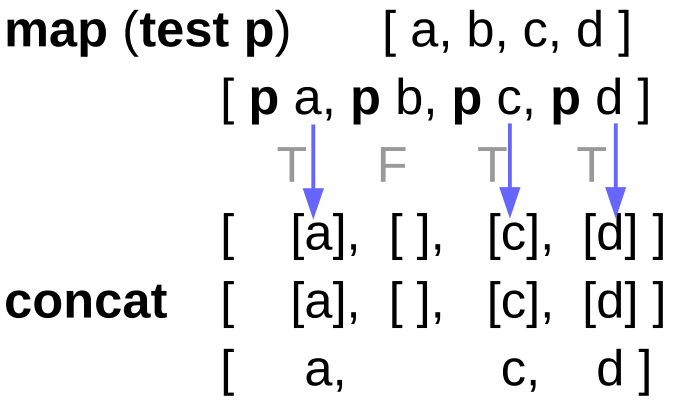


# Definitions of filter

```
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**

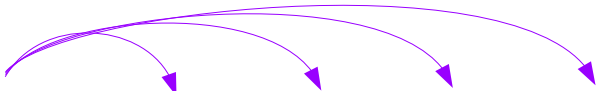
**map f . tail** [ a , b, c, d] =
**map f** [ b, c, d ] = [ **f** b, **f** c, **f** d ]
**tail** . **map f** [ a, b, c, d] =
**tail** [ **f** a, **f** b, **f** c, **f** d ] = [ **f** b, **f** c, **f** d ]

**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
```

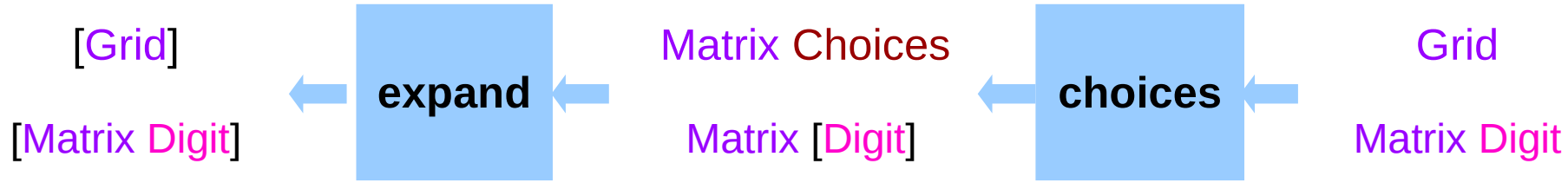
```
filter (p . f) . map f = map f . filter p
```

# Solve with pruning

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

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

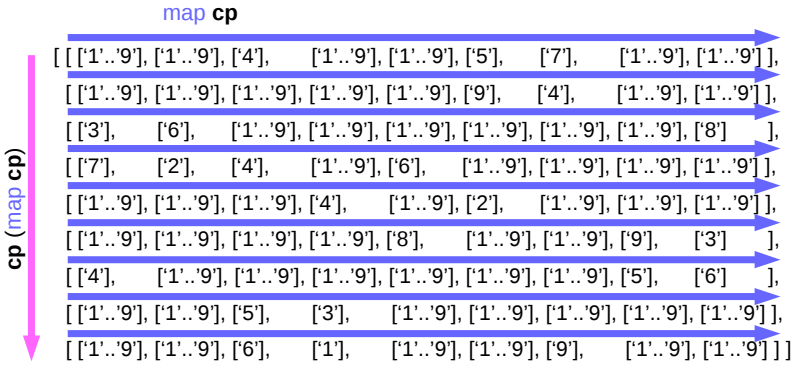
**prune** :: Matrix [Digit] -> Matrix [Digit]



# expand – map cp

Matrix Choices

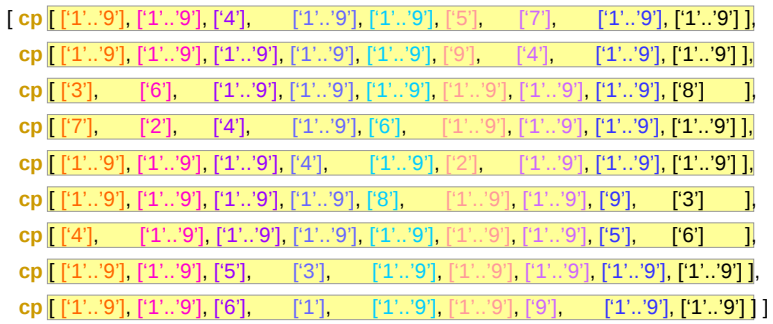
Matrix [Digit]

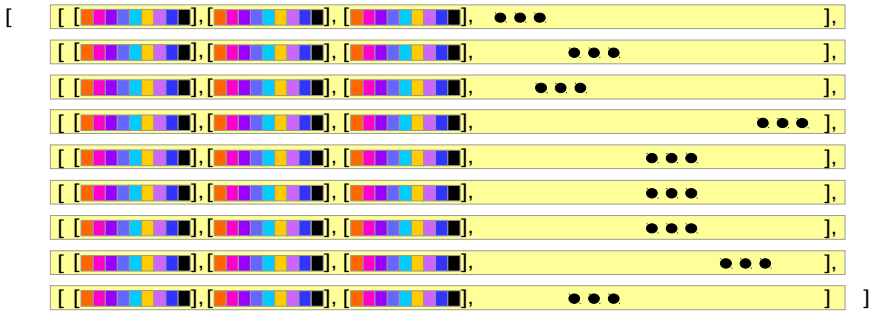



**expand** :: Matrix Choices -> [Grid]

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

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

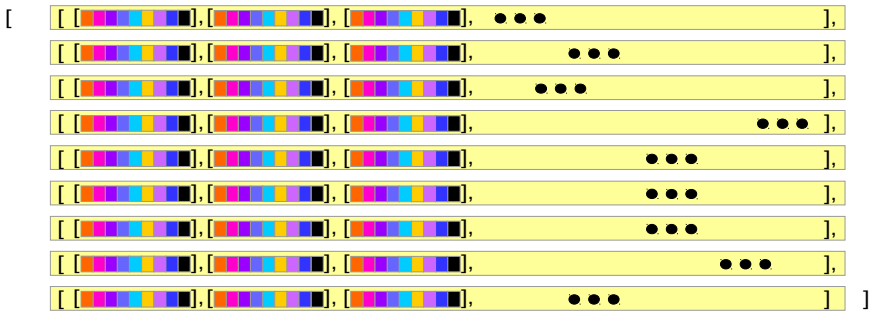


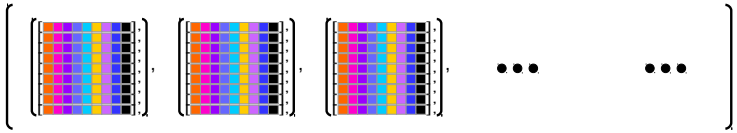
map **cp**

# expand – cp . map cp

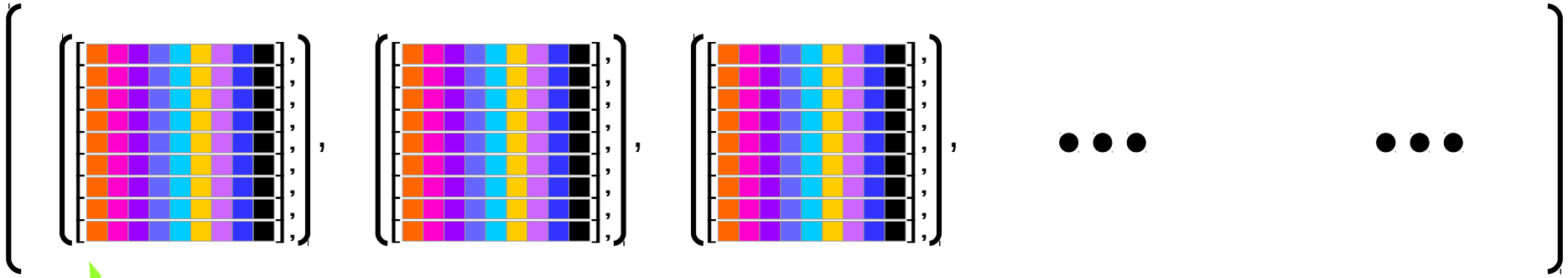
map **cp**



**cp** . map **cp**



[Grid]
[Matrix Digit]

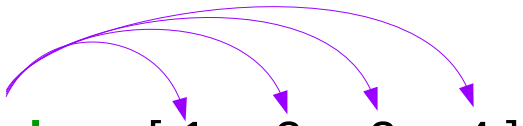


g

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

# Valid

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

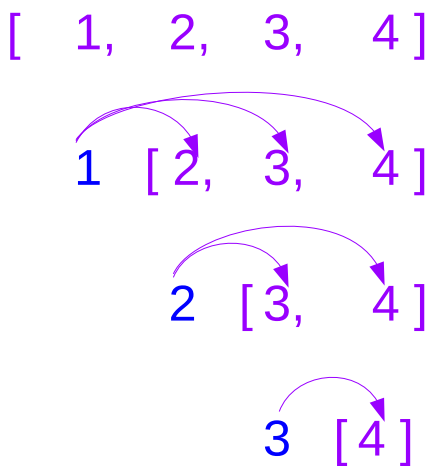


```
all nodups [r1   r2   r3   r4 ]
```

```
(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

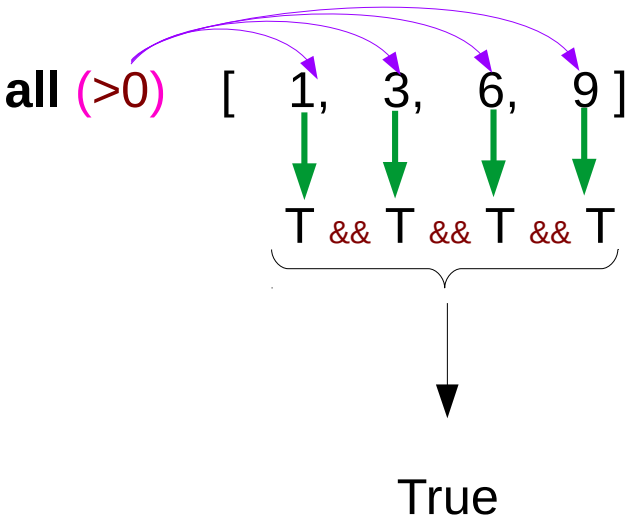
```
nodups :: Eq a => [a] -> Bool
nodups []       = True
nodups (x:xs) = x `notElem` xs && nodups xs

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
```

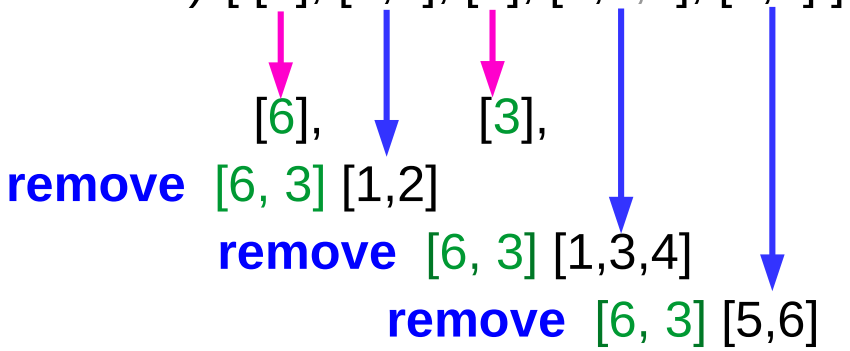
```
pruneRow [ [6], [1,2], [3], [1,3,4], [5,6] ]      -- Fixed →  [6, 3]
[[6], [1,2], [3], [1,4], [5]]

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

```
boxs  . 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
```

# map and filter

Assume **f . f = id**

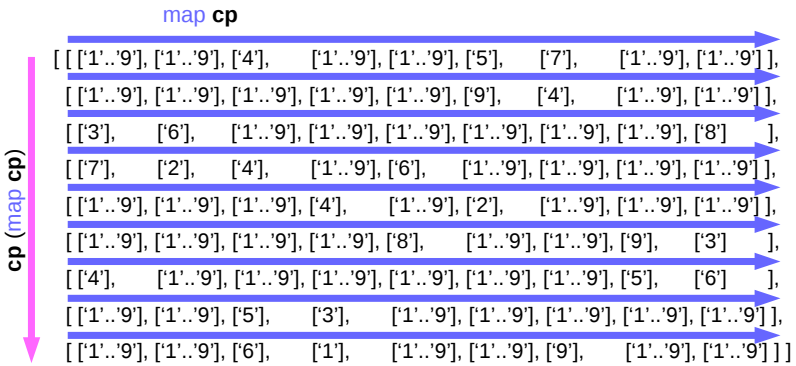
**filter (p . f) = map f . filter p . map f**

filter by **p.f**

first apply **f**, then filter by **p**
filtered element : applied by **f**
double application **f** : back to original element

```
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 = cp . map cp




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

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

Col cp . Row cp

**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 . Boxes**

first rearrange box elements into rows, then expand
first expand, then rearrange box elements into rows
over the each of the expansion

# expand and boxs

For the row-wise access,
**cols** rearrange column elements into rows
**boxs** rearrange box elements into rows

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

```
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 and cp

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

```
A                 =    [[1, 2, 3], [2], [1, 3]]
cp A              =    [[1, 2, 1], [1, 2, 3], [2, 2, 1], [2, 2, 3], [3, 2, 1], [3, 2, 3]]
```

```
filter (all p) . cp A  = filter [ all p [1, 2, 1], all p [1, 2, 3], all p [2, 2, 1],
                                  all p [2, 2, 3], all p [3, 2, 1], all p [3, 2, 3] ]

                       = filter [ and [p 1, p 2, p 1], and [p 1, p 2, p 3], and [p 2, p 2, p 1],
                                  and [p 2, p 2, p 3], and [p 3, p 2, p 1], and [p 3, p 2, p 3] ]
```

```
map (filter p) A  =    [ filter p [1, 2, 3], filter p [2], filter p [1, 3] ]
                  =    [ filter [p 1, p 2, p 3], [p 2], [p 1, p 3] ]

cp . map (filter p) A   =  cp . [ filter [p 1, p 2, p 3], [p 2], [p 1, p 3] ]
```

# map and filter

**filter (all p) . cp = cp . map (filter p)**

First **cp**, then filter by **all.p** – every element of every list must satisfy **p**
First filter by **p** then **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 composition

```
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) . Boxes
```

```
map (filter nodups) . map cp     =     map (filter nodups . cp)
```

# PruneRow property

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

```
filter nodups . cp = filter nodups . cp . pruneRow
```

```
filter valid . expand = filter valid . expand . prune
prune = pruneBy boxs . pruneBy cols . pruneBy rows
  where pruneBy f = f . map pruneRow . f
pruneRow row = map (remove ones) row
   where ones = [d | [d] <- row]
```

# PruneRow property

```
  filter (all nodups . boxs) . expand
```

```
          *    *    *
```

```
= map boxs . cp . map (filter nodups . cp) . Boxes
= map boxs . cp . map (filter nodups . cp . pruneRow) . Boxes
```

```
          *    *    *
```

```
= filter (all nodups . boxs) . expand . pruneBy boxs
```

# prunBy boxs  proof – (1)

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

# prunBy boxs  proof – (2)

**= map boxs . cp . map (filter nodups . cp . pruneRow) . boxs**

**= map boxs . cp . map (filter nodups) . map (cp . pruneRow) . boxs**

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

**= map boxs . cp . map (filter nodups) . map (cp . pruneRow) . boxs**

**= map boxs . filter (all nodups) . cp . map (cp . pruneRow) . boxs**

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

**= map boxs . filter (all nodups) . cp . map (cp . pruneRow) . boxs**

**= map boxs . filter (all nodups) . cp . map cp . map pruneRow . boxs**

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

**= map boxs . filter (all nodups) . cp . map cp . map pruneRow . boxs**

**= map boxs . filter (all nodups) . expand . map pruneRow . boxs**

expand = cp . map cp

# prunBy boxs  proof – (3)

= map boxs . filter (all nodups) . expand . map pruneRow . boxs
= filter (all nodups . boxs) . map boxs . expand . map pruneRow . boxs

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

= filter (all nodups . boxs) . map boxs . expand . map pruneRow . boxs
= filter (all nodups . boxs) . expand . boxs . map pruneRow . boxs

map  boxs . expand = expand . boxs

= filter (all nodups . boxs) . expand . boxs . map pruneRow . boxs
= filter (all nodups . boxs) . expand . pruneBy boxs

pruneBy f = f . map pruneRow . f

# Pruning (1)

```
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
```

# Pruning (2)

**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**

# Pruning (3)

**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**

# Pruning (4)

**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**

# Pruning (5)

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

## References

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