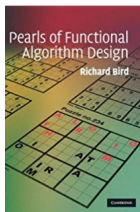


Coding dojo
A simple Sudoku solver in Haskell

B^oFP

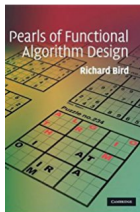
Bucharest FP #027

Welcome



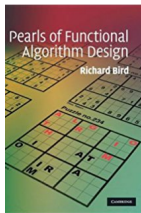
- Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)

Welcome

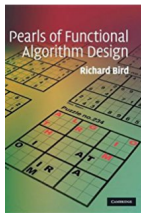


- ▶ Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)
- ▶ Twelve short functions:
 - ▶ 5 easy (★)
 - ▶ 4 medium (★★)
 - ▶ 3 challenging (★★★)

Welcome

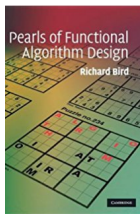


- ▶ Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)
- ▶ Twelve short functions:
 - ▶ 5 easy (★)
 - ▶ 4 medium (★★)
 - ▶ 3 challenging (★★★)
- ▶ Programming techniques:
 - ▶ Top-down programming / wishful thinking

[illegible]

- 2 / 21

Welcome



- ▶ Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)
- ▶ Twelve short functions:
 - ▶ 5 easy (★)
 - ▶ 4 medium (★★)
 - ▶ 3 challenging (★★★)
- ▶ Programming techniques:
 - ▶ Top-down programming / wishful thinking
 - ▶ Wholemeal programming (prevents a disease called “indexitis”)
 - ▶ Higher-order functions, recursion, point-free style

How to play Sudoku

	2	4	
1			3
4			2
	1	3	

$N = 2$

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

$N = 3$

Fill in the empty cells with digits 1 to N^2 such that every row, column and $N \times N$ box contains the digits 1 to N^2 .

How to play Sudoku

3	2	4	1
1	4	2	3
4	3	1	2
2	1	3	4

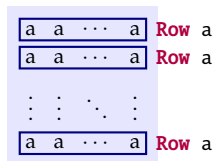
$N = 2$

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
9	7	1	3	8	5	6	2	4
5	4	3	7	2	6	8	1	9
6	8	2	1	4	9	7	5	3
7	9	4	6	3	2	5	8	1
2	6	5	8	1	4	9	3	7
3	1	8	9	5	7	4	6	2

$N = 3$

Fill in the empty cells with digits 1 to N^2 such that every row, column and $N \times N$ box contains the digits 1 to N^2 .

Data types

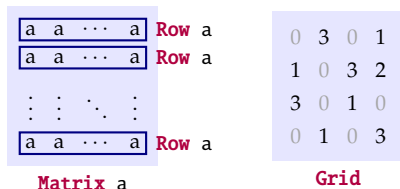


Matrix a

type **Matrix** a = [**Row** a]

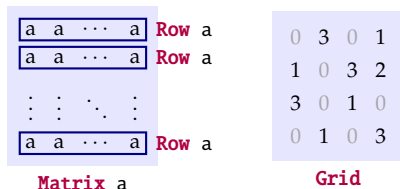
type **Row** a = [a]

Data types



```
type Matrix a = [Row a]
type Row a = [a]
type Grid = Matrix Digit
type Digit = Int
```

Data types



```
type Matrix a = [Row a]
type Row a = [a]
type Grid = Matrix Digit
type Digit = Int
```

We assume that digit zero indicates an empty cell:

```
isEmpty :: Digit -> Bool
isEmpty 0 = True
isEmpty _ = False
```

Exercise 1: solve [★]

```
solve :: Grid -> [Grid]
solve = undefined
```

Exercise 1: solve [★]

```
solve :: Grid -> [Grid]
solve = undefined
```

Given:

```
-- Generates grids by replacing empty entries
-- with all possible choices
completions :: Grid -> [Grid]
```

Exercise 1: solve [★]

```
solve :: Grid -> [Grid]
solve = undefined
```

Given:

```
-- Generates grids by replacing empty entries
-- with all possible choices
completions :: Grid -> [Grid]

-- Tests whether a grid is a valid solution:
-- has different entries in each row, column and box
valid :: Grid -> Bool
```

Exercise 1: solve [★]

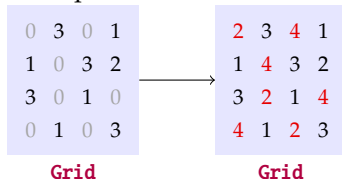
```
solve :: Grid -> [Grid]
solve = undefined
```

Given:

```
-- Generates grids by replacing empty entries
-- with all possible choices
completions :: Grid -> [Grid]

-- Tests whether a grid is a valid solution:
-- has different entries in each row, column and box
valid :: Grid -> Bool
```

Example:



Exercise 2: completions [★]

```
completions :: Grid -> [Grid]
completions = undefined
```


Exercise 2: completions [★]

```
completions :: Grid -> [Grid]
completions = undefined
```

Given:

```
-- Replaces empty entries with all possible choices
-- for that entry
choices :: Grid -> Matrix [Digit]
```

Exercise 2: completions [★]

```
completions :: Grid -> [Grid]
completions = undefined
```

Given:

```
-- Replaces empty entries with all possible choices
-- for that entry
choices :: Grid -> Matrix [Digit]

-- Generates a list of all possible boards
-- from a given matrix of choices
expand :: Matrix [Digit] -> [Grid]
```

Exercise 2: completions [★]

```
completions :: Grid -> [Grid]
completions = undefined
```

Given:

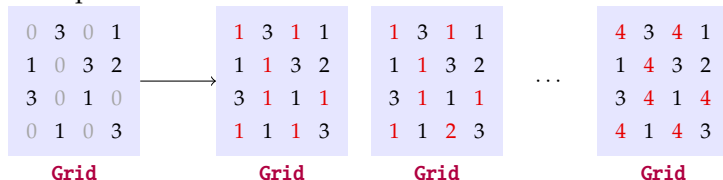
```
-- Replaces empty entries with all possible choices
-- for that entry
```

```
choices :: Grid -> Matrix [Digit]
```

```
-- Generates a list of all possible boards
-- from a given matrix of choices
```

```
expand :: Matrix [Digit] -> [Grid]
```

Example:



Exercise 3: choices [★]

```
choices :: Grid -> Matrix [Digit]  
choices = undefined
```

Exercise 3: choices [★]

```
choices :: Grid -> Matrix [Digit]
```

```
choices = undefined
```

Example:

0	3	0	1
1	0	3	2
3	0	1	0
0	1	0	3

Matrix Digit



1 2 3 4	3	1 2 3 4	1
1	1 2 3 4	3	2
3	1 2 3 4	1	1 2 3 4
1 2 3 4	1	1 2 3 4	3

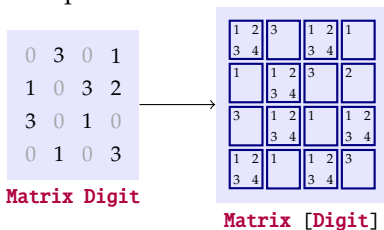
Matrix [Digit]

Exercise 3: choices [★]

```
choices :: Grid -> Matrix [Digit]
```

```
choices = undefined
```

Example:



Hint:

- ▶ Define a helper function `choice :: Digit -> [Digit]`

Exercise 4: expand [**]

```
expand :: Matrix [Digit] -> [Grid]  
expand = undefined
```

Exercise 4: expand [★★]

```
expand :: Matrix [Digit] -> [Grid]
expand = undefined
```

Given:

```
-- Computes the cartesian product of a list of lists
cp :: [[a]] -> [[a]]
```

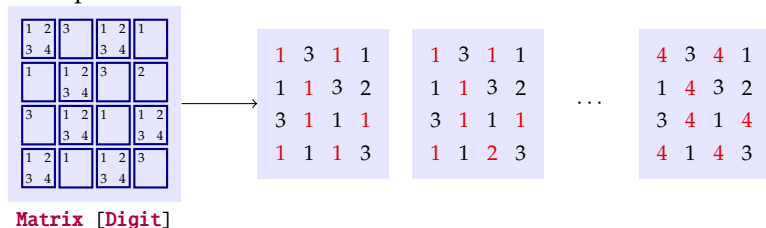

Exercise 4: expand [**]

```
expand :: Matrix [Digit] -> [Grid]
expand = undefined
```

Given:

```
-- Computes the cartesian product of a list of lists
cp :: [[a]] -> [[a]]
```

Example:



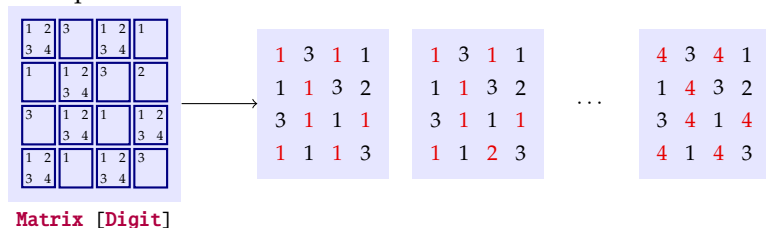
Exercise 4: expand [**]

```
expand :: Matrix [Digit] -> [Grid]
expand = undefined
```

Given:

```
-- Computes the cartesian product of a list of lists
cp :: [[a]] -> [[a]]
```

Example:



Hints:

- ▶ First generate all combinations across each row
- ▶ Then combine those generated combinations

Exercise 5: `cp` $[\star \star \star]$

```
cp :: [[a]] -> [[a]]  
cp = undefined
```

Exercise 5: $\text{cp} \left[\begin{smallmatrix} \star & \star & \star \end{smallmatrix} \right]$

```
cp :: [[a]] -> [[a]]
```

```
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Exercise 5: `cp` `[***]`

```
cp :: [[a]] -> [[a]]  
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion:

Exercise 5: cp [***]

```
cp :: [[a]] -> [[a]]  
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion:

xss



Exercise 5: cp [***]

```
cp :: [[a]] -> [[a]]
```

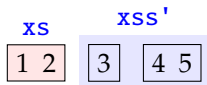
```
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion:



Exercise 5: cp [***]

```
cp :: [[a]] -> [[a]]
```

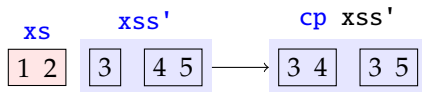
```
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion:



Exercise 5: $\text{cp} \llbracket \star \star \star \rrbracket$

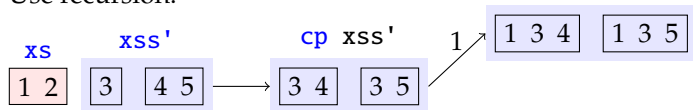
```
cp :: [[a]] -> [[a]]  
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion:



Exercise 5: $\text{cp} \llbracket \star \star \star \rrbracket$

```
cp :: [[a]] -> [[a]]
```

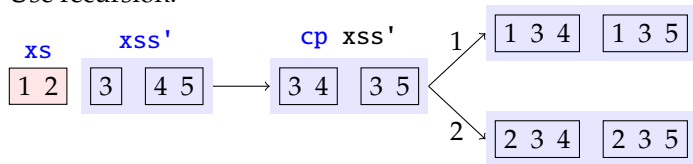
```
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion:



Exercise 5: $cp \ [***]$

```
cp :: [[a]] -> [[a]]
```

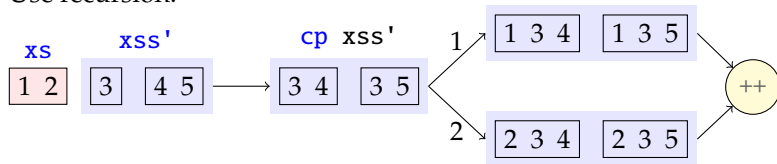
```
cp = undefined
```

Example:

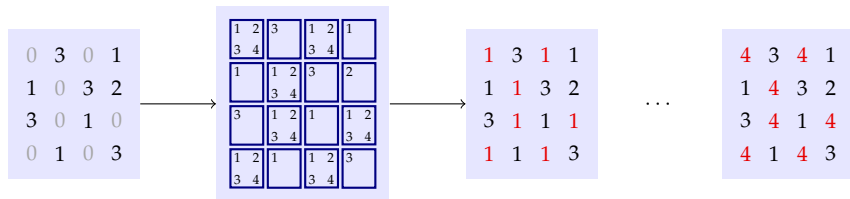
```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion:

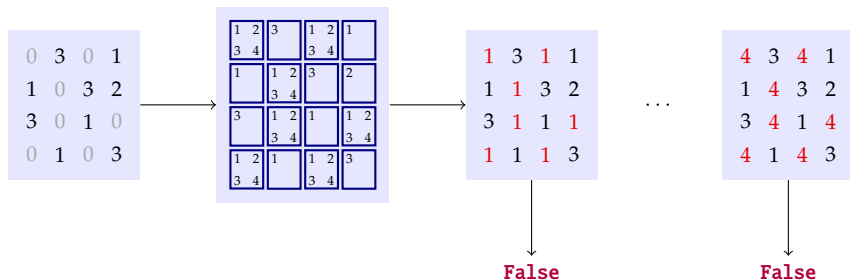


Intermezzo



- Finished implementing the **completions** function

Intermezzo



- ▶ Finished implementing the **completions** function
- ▶ Next, the **valid** function: test whether a grid is a valid solution

Exercise 6: valid [★★]

```
valid :: Grid -> Bool  
valid = undefined
```

Exercise 6: valid [★★]

```
valid :: Grid -> Bool  
valid = undefined
```

Given:

```
-- Checks that a list contains no duplicates  
nodups :: [a] -> Bool
```

Exercise 6: valid [★★]

```
valid :: Grid -> Bool
valid = undefined
```

Given:

```
-- Checks that a list contains no duplicates
nodups :: [a] -> Bool

-- Re-orders the values from a matrix's rows, columns
-- or boxes to appear along the rows
rows :: Matrix a -> Matrix a
cols :: Matrix a -> Matrix a
boxs :: Matrix a -> Matrix a
```


Exercise 6: valid [★★]

```
valid :: Grid -> Bool
```

```
valid = undefined
```

Given:

```
-- Checks that a list contains no duplicates
```

```
nodups :: [a] -> Bool
```

```
-- Re-orders the values from a matrix's rows, columns
```

```
-- or boxes to appear along the rows
```

```
rows :: Matrix a -> Matrix a
```

```
cols :: Matrix a -> Matrix a
```

```
boxes :: Matrix a -> Matrix a
```

Examples:

1	3	1	1
1	1	3	2
3	1	1	1
1	1	1	3

→ False

2	3	4	1
1	4	3	2
3	2	1	4
4	1	2	3

→ True

Exercise 7: nodups [★★]

```
nodups :: [a] -> Bool  
nodups = undefined
```

Exercise 7: nodups [★★]

```
nodups :: [a] -> Bool
nodups = undefined
```

Examples:

```
nodups [] = True
nodups [1, 2, 3] = True
nodups [1, 2, 1] = False
```

Exercise 7: nodups [★★]

```
nodups :: [a] -> Bool
nodups = undefined
```

Examples:

```
nodups [] = True
nodups [1, 2, 3] = True
nodups [1, 2, 1] = False
```

Hints:

- ▶ Use recursion
- ▶ Use [Hoogle](#) to find a function of type `a -> [a] -> Bool`

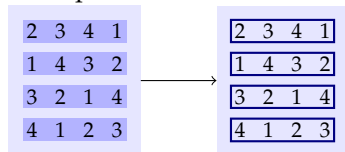
Exercise 8: rows [★]

```
rows :: Matrix a -> Matrix a
```

Exercise 8: rows [★]

```
rows :: Matrix a -> Matrix a
```

Example:



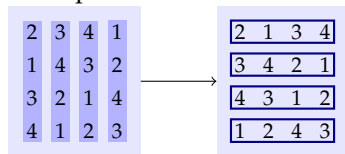
Exercise 9: cols [***]

```
cols :: Matrix a -> Matrix a
```

Exercise 9: cols [***]

```
cols :: Matrix a -> Matrix a
```

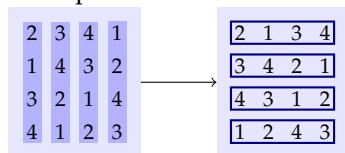
Example:



Exercise 9: cols [***]

`cols :: Matrix a -> Matrix a`

Example:



Hints:

- ▶ Use recursion
- ▶ Define a case for a one-row matrix; example:
`cols [[1,2,3,4]] = [[1],[2],[3],[4]]`
- ▶ For the recursive case, use the `zipWith` function:
`zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

Exercise 10: boxs $[\star \star \star]$

```
boxs :: Matrix a -> Matrix a
```

Exercise 10: boxs [***]

```
boxs :: Matrix a -> Matrix a
```

Given:

```
-- Groups a list into lists of length two
```

```
group :: [a] -> [[a]]
```

Exercise 10: boxs [***]

```
boxs :: Matrix a -> Matrix a
```

Given:

```
-- Groups a list into lists of length two
```

```
group :: [a] -> [[a]]
```

```
-- Flattens a nested list of elements
```

```
ungroup :: [[a]] -> [a]
```

Exercise 10: boxs [***]

```
boxs :: Matrix a -> Matrix a
```

Given:

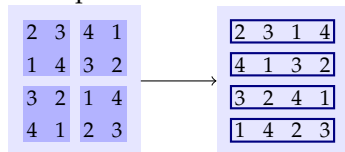
```
-- Groups a list into lists of length two
```

```
group :: [a] -> [[a]]
```

```
-- Flattens a nested list of elements
```

```
ungroup :: [[a]] -> [a]
```

Example:



Exercise 10: boxs [***]

```
boxs :: Matrix a -> Matrix a
```

Given:

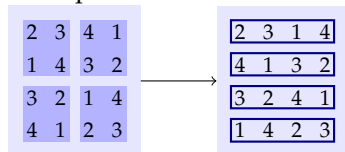
```
-- Groups a list into lists of length two
```

```
group :: [a] -> [[a]]
```

```
-- Flattens a nested list of elements
```

```
ungroup :: [[a]] -> [a]
```

Example:



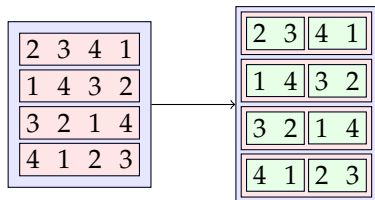
Hints:

- ▶ Use the previously defined `cols` function
- ▶ Chain five transformations (see next slide)

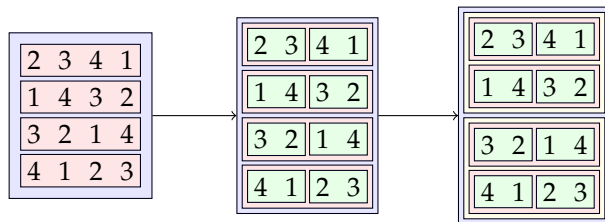
Exercise 10: boxes $[\star \star \star]$

2	3	4	1
1	4	3	2
3	2	1	4
4	1	2	3

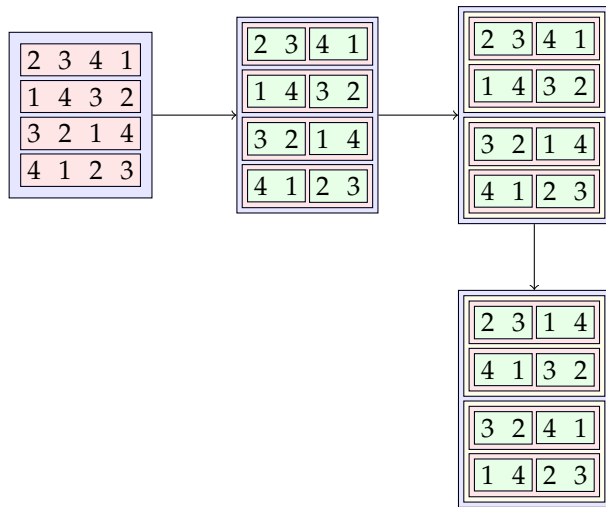
Exercise 10: boxes $[\star \star \star]$



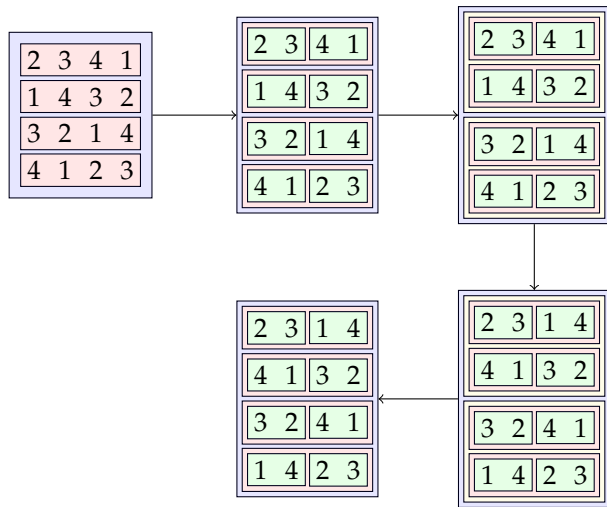
Exercise 10: boxes $[\star \star \star]$



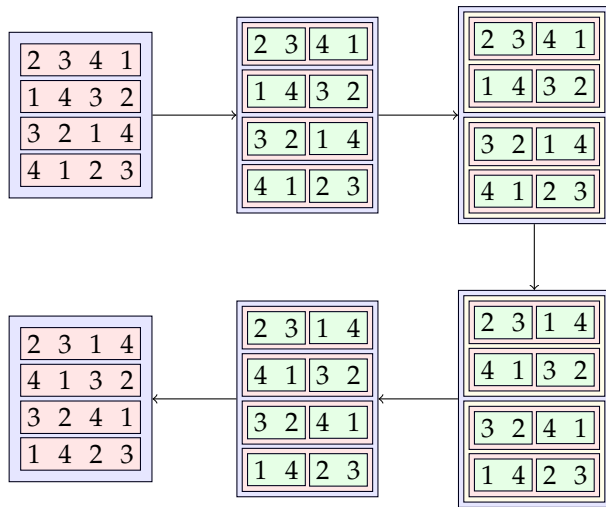
Exercise 10: boxs [***]



Exercise 10: boxes $[\star\star\star]$



Exercise 10: boxes $[\star\star\star]$



Exercise 11: group [★★]

```
group :: [a] -> [[a]]
```

Exercise 11: group [★★]

`group :: [a] -> [[a]]`

Example:

`group [1,2,3,4] = [[1,2],[3,4]]`

Exercise 12: ungroup [★]

```
ungroup :: [[a]] -> [a]
```

Exercise 12: ungroup [★]

`ungroup :: [[a]] -> [a]`

Example:

`ungroup [[1,2],[3,4]] = [1,2,3,4]`

Exercise 12: ungroup [★]

`ungroup :: [[a]] -> [a]`

Example:

`ungroup [[1,2],[3,4]] = [1,2,3,4]`

Hints:

- ▶ Use [Hoogle](#)

That's all folks

- ▶ Time to solve some Sudokus

That's all folks

- ▶ Time to solve some Sudokus
- ▶ The current approach is inefficient, but correct

That's all folks

- ▶ Time to solve some Sudokus
- ▶ The current approach is inefficient, but correct
- ▶ Equational reasoning to improve performance

That's all folks

- ▶ Time to solve some Sudokus
 - ▶ The current approach is inefficient, but correct
 - ▶ Equational reasoning to improve performance
 - ▶ Define a function `prune` that eliminates early invalid solutions
- `filter valid . expand = filter valid . expand . prune`

That's all folks

- ▶ Time to solve some Sudokus
- ▶ The current approach is inefficient, but correct
- ▶ Equational reasoning to improve performance
 - ▶ Define a function `prune` that eliminates early invalid solutions
`filter valid . expand = filter valid . expand . prune`
 - ▶ It is not hard to define a function to prune a row (exercise):
`pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]`

That's all folks

- ▶ Time to solve some Sudokus
- ▶ The current approach is inefficient, but correct
- ▶ Equational reasoning to improve performance
 - ▶ Define a function `prune` that eliminates early invalid solutions
`filter valid . expand = filter valid . expand . prune`
 - ▶ It is not hard to define a function to prune a row (exercise):
`pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]`
 - ▶ Equational reasoning to define `prune` in terms of `pruneRow`

That's all folks

- ▶ Time to solve some Sudokus
- ▶ The current approach is inefficient, but correct
- ▶ Equational reasoning to improve performance
 - ▶ Define a function `prune` that eliminates early invalid solutions
`filter valid . expand = filter valid . expand . prune`
 - ▶ It is not hard to define a function to prune a row (exercise):
`pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]`
 - ▶ Equational reasoning to define `prune` in terms of `pruneRow`
 - ▶ The function `pruneRow` satisfies the equation
`filter nodups . cp = filter nodups . cp . pruneRow`

That's all folks

- ▶ Time to solve some Sudokus
- ▶ The current approach is inefficient, but correct
- ▶ Equational reasoning to improve performance
 - ▶ Define a function `prune` that eliminates early invalid solutions
`filter valid . expand = filter valid . expand . prune`
 - ▶ It is not hard to define a function to prune a row (exercise):
`pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]`
 - ▶ Equational reasoning to define `prune` in terms of `pruneRow`
 - ▶ The function `pruneRow` satisfies the equation
`filter nodups . cp = filter nodups . cp . pruneRow`
 - ▶ Expand the expression `filter valid . expand`

That's all folks

- ▶ Time to solve some Sudokus
- ▶ The current approach is inefficient, but correct
- ▶ Equational reasoning to improve performance
 - ▶ Define a function `prune` that eliminates early invalid solutions
`filter valid . expand = filter valid . expand . prune`
 - ▶ It is not hard to define a function to prune a row (exercise):
`pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]`
 - ▶ Equational reasoning to define `prune` in terms of `pruneRow`
 - ▶ The function `pruneRow` satisfies the equation
`filter nodups . cp = filter nodups . cp . pruneRow`
 - ▶ Expand the expression `filter valid . expand`
 - ▶ Use the above equation and compress back the formula

Further references

- ▶ Richard Bird's papers and books (Bird, 2006, 2010, 2014)
- ▶ Conor McBride's Sudoku solver using applicative and traversable:
<https://stackoverflow.com/a/10242673/474311>

References

- Bird, R. (2010). *Pearls of Functional Algorithm Design*. Cambridge University Press.
- Bird, R. (2014). *Thinking Functionally with Haskell*. Cambridge University Press.
- Bird, R. S. (2006). A program to solve Sudoku. *Journal of Functional Programming*, 16(6):671–679.