Programming in Haskell: Lecture 26

S P Suresh

November 13, 2019

Sudoku

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

Basic structures

- Basic data structures:

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

Basic structures

- Basic data structures:

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

- Choices for each cell:

```
type Choices = [Digit]
```

Basic structures

- Basic data structures:

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

- Choices for each cell:

```
type Choices = [Digit]
```

- Grid entries:

```
digits = "123456789"
blank :: Digit -> Bool
blank = (== '-')
```

High-level strategy

- Generate the list of all solutions and pick the first one

High-level strategy

- Generate the list of all solutions and pick the first one
  - Hopefully there is only one solution

High-level strategy

- Generate the list of all solutions and pick the first one
  - Hopefully there is only one solution
- Expand the given grid to all possible valid complete grids

High-level strategy

- Generate the list of all solutions and pick the first one
  - Hopefully there is only one solution
- Expand the given grid to all possible valid complete grids
  - Fill in each empty cell with all choices

High-level strategy

- Generate the list of all solutions and pick the first one
  - Hopefully there is only one solution
- Expand the given grid to all possible valid complete grids
  - Fill in each empty cell with all choices
  - Expand a matrix of choices to a list of complete grids

High-level strategy

- Generate the list of all solutions and pick the first one
  - Hopefully there is only one solution
- Expand the given grid to all possible valid complete grids
  - Fill in each empty cell with all choices
  - Expand a matrix of choices to a list of complete grids
  - Choose all valid grids from this list

High-level strategy

- Generate the list of all solutions and pick the first one
  - Hopefully there is only one solution
- Expand the given grid to all possible valid complete grids
  - Fill in each empty cell with all choices
  - Expand a matrix of choices to a list of complete grids
  - Choose all valid grids from this list
- `solve` implements this strategy:

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

Filling in all choices

- Filling a cell with choices:

```
choice :: Digit -> [Digit]
choice d = if blank d then digits else [d]
```

Filling in all choices

- Filling a cell with choices:

```
choice :: Digit -> [Digit]
choice d = if blank d then digits else [d]
```

- `map choice` fills all cells in a row with choices

Filling in all choices

- Filling a cell with choices:

```
choice :: Digit -> [Digit]
choice d = if blank d then digits else [d]
```

- `map choice` fills all cells in a row with choices
- To fill all cells in a grid with choices:

```
choices :: Grid -> Matrix Choices
choices = map (map choice)
```

Expanding list of choices

- We take cartesian product of the matrix of choices using:

```
cp :: [[a]] -> [[a]]
cp []         = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

Expanding list of choices

- We take cartesian product of the matrix of choices using:

```
cp :: [[a]] -> [[a]]
cp []         = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

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

Expanding list of choices

- We take cartesian product of the matrix of choices using:

```
    cp :: [[a]] -> [[a]]
    cp []         = [[]]
    cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

- `cp [[1,2], [3,4]] = [[1,3], [1,4], [2,3], [2,4]]`
- `expand` computes the list of all complete grids:

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

Valid grids

- In a valid complete grid:

Valid grids

- In a valid complete grid:
  - Each row has distinct entries

Valid grids

- In a valid complete grid:
  - Each row has distinct entries
  - Each column has distinct entries

Valid grids

- In a valid complete grid:
  - Each row has distinct entries
  - Each column has distinct entries
  - Each 3×3 box has distinct entries

Valid grids

- In a valid complete grid:
  - Each row has distinct entries
  - Each column has distinct entries
  - Each 3×3 box has distinct entries
- Checking for distinct entries in a list:

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

Valid grids

- In a valid complete grid:
  - Each row has distinct entries
  - Each column has distinct entries
  - Each 3×3 box has distinct entries
- Checking for distinct entries in a list:

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

- `all` is a built-in function:

```
all p []     = True
all p (x:xs) = p x && all p xs
```

Valid grids

- A grid is a list of 9 rows

Valid grids

- A grid is a list of 9 rows
  - Each row is a list of 9 digits

Valid grids

- A grid is a list of 9 rows
  - Each row is a list of 9 digits
- Extracting all rows of a grid:

```
rows = id
```

Valid grids

- A grid is a list of 9 rows
  - Each row is a list of 9 digits
- Extracting all rows of a grid:

```
rows = id
```

- Extracting all columns:

```
cols [xs]       = [[x] | x <- xs]
cols (xs:xss) = zipWith (:) xs (cols xss)
```

Valid grids

- A grid is a list of 9 rows
  - Each row is a list of 9 digits
- Extracting all rows of a grid:

```
rows = id
```

- Extracting all columns:

```
cols [xs]     = [[x] | x <- xs]
cols (xs:xss) = zipWith (:) xs (cols xss)
```

- `cols [[1,2], [3,4], [5,6]] = [[1,3,5], [2,4,6]]`

- Extracting the 3×3 boxes:

```
boxs :: Matrix a -> Matrix a
boxs = map ungroup . ungroup . map cols .
                               group . map group
group :: [a] -> [[a]]
group [] = []
group xs = take 3 xs:group (drop 3 xs)
ungroup :: [[a]] -> [a]
ungroup = concat
```

Illustrating boxs *on* 4×4

$$\begin{matrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{matrix}$$

Illustrating `boxs` *on* 4×4

$$\begin{matrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{matrix}$$

map group

Illustrating boxs on 4×4

$$\begin{matrix} ab & cd \\ ef & gh \\ ij & kl \\ mn & op \end{matrix}$$

Illustrating boxs *on* 4×4

$$\begin{array}{cc} ab & cd \\ ef & gh \\ ij & kl \\ mn & op \end{array}$$

group

Illustrating boxs *on* 4×4

$$\begin{matrix} ab & cd \\ ef & gh \\ \\ ij & kl \\ mn & op \end{matrix}$$

Illustrating `boxs` *on* 4×4

$$\begin{array}{cc} ab & cd \\ ef & gh \\ \\ ij & kl \\ mn & op \end{array}$$

`map cols`

Illustrating boxs on 4×4

$$\begin{matrix} ab & ef \\ cd & gh \\ \\ ij & mn \\ kl & op \end{matrix}$$

Illustrating boxs *on* 4×4

$$\begin{matrix} ab & ef \\ cd & gh \end{matrix}$$

$$\begin{matrix} ij & mn \\ kl & op \end{matrix}$$

ungroup

Illustrating boxs *on* 4×4

$$\begin{array}{cc} ab & ef \\ cd & gh \\ ij & mn \\ kl & op \end{array}$$

Illustrating `boxs` on 4×4

$$\begin{array}{cc} ab & ef \\ cd & gh \\ ij & mn \\ kl & op \end{array}$$

map ungroup

Illustrating boxs on 4×4

$$\begin{matrix} a & b & e & f \\ c & d & g & h \\ i & j & m & n \\ k & l & o & p \end{matrix}$$

Illustrating boxs on 4×4

$$\begin{vmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{vmatrix} \longrightarrow \begin{vmatrix} a & b & e & f \\ c & d & g & h \\ i & j & m & n \\ k & l & o & p \end{vmatrix}$$

All valid solutions

```
boxs :: Matrix a -> Matrix a
boxs = map ungroup . ungroup . map cols . group . map group

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

solve = filter valid . expand . choices
solution = head . solve
```

All valid solutions

```
puzzle :: Grid
puzzle = [ "--4--57--"
         , "-----94--"
         , "36------8"
         , "72--6----"
         , "---4-2---"
         , "----8--93"
         , "4------56"
         , "--53-----"
         , "--61--9--"
         ]
```

Sudoku example

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

`puzzle`

Sudoku example

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

solution puzzle

Remarks on the program

- Our program is useless

Remarks on the program

- Our program is useless
- Even with half the grid filled, we have to check 9^{40} grids for validity

Remarks on the program

- Our program is useless
- Even with half the grid filled, we have to check 9^{40} grids for validity
    - $9^{40} = 147808829414345923316083210206383297601$

Remarks on the program

- Our program is useless
- Even with half the grid filled, we have to check 9^{40} grids for validity
  - $9^{40} = 147808829414345923316083210206383297601$
- Takes forever even for a grid with only 9 blank cells

```
solution ["-52439817", "8-9165432", "41-872596",
          "548-97321", "9315-4768", "26738-945",
          "795213-84", "1849562-3", "32674815-"]


["652439817","879165432","413872596"
,"548697321","931524768","267381945"
,"795213684","184956273","326748159"]
(946.02 secs, 703,822,023,848 bytes)
```

A better strategy?

- **Obvious improvement**: Try to prune the choices even before expanding to a list of grids

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

A better strategy?

- **Obvious improvement**: Try to prune the choices even before expanding to a list of grids

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

- We would like `prune` to satisfy:

```
filter valid . expand . prune  = filter valid . expand
```

Pruning the choices

- How do we prune the choices?

Pruning the choices

- How do we prune the choices?
- Consider a row with three entries and six blank cells

Pruning the choices

- How do we prune the choices?
- Consider a row with three entries and six blank cells
- Let the entries be 1, 5 and 9

Pruning the choices

- How do we prune the choices?
- Consider a row with three entries and six blank cells
- Let the entries be 1, 5 and 9
- Then the list of choices for the blank cells is [234678]

Pruning the choices

- How do we prune the choices?
- Consider a row with three entries and six blank cells
- Let the entries be `1`, `5` and `9`
- Then the list of choices for the blank cells is `[234678]`
- If some column has entries `1`, `4` and `7`, choices further pruned to `[2368]`

Pruning the choices

- How do we prune the choices?
- Consider a row with three entries and six blank cells
- Let the entries be `1`, `5` and `9`
- Then the list of choices for the blank cells is `[234678]`
- If some column has entries `1`, `4` and `7`, choices further pruned to `[2368]`
- Similar pruning based on entries in 3×3 box

Pruning the choices

- How do we prune the choices?
- Consider a row with three entries and six blank cells
- Let the entries be `1`, `5` and `9`
- Then the list of choices for the blank cells is `[234678]`
- If some column has entries `1`, `4` and `7`, choices further pruned to `[2368]`
- Similar pruning based on entries in 3×3 box
- Potentially huge savings!

Pruning the choices

- Note that:

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

Pruning the choices

- Note that:

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

- To prune based on boxes, apply `boxs`

Pruning the choices

- Note that:

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

- To prune based on boxes, apply `boxs`
- Now each row is a box of the original grid

Pruning the choices

- Note that:

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

- To prune based on boxes, apply `boxs`
- Now each row is a box of the original grid
- Prune each row

Pruning the choices

- Note that:

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

- To prune based on boxes, apply `boxs`
- Now each row is a box of the original grid
- Prune each row
- Apply `boxs` again to restore order of cells

Pruning the choices

- Note that:

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

- To prune based on boxes, apply `boxs`
- Now each row is a box of the original grid
- Prune each row
- Apply `boxs` again to restore order of cells
- Similarly with `cols`

Pruning the choices

```
prune             = pruneBy boxs .
                    pruneBy cols . pruneBy rows
pruneBy f         = f . map pruneRow . f

pruneRow row      = map (remove fixed) row
    where fixed   = [d | [d] <- row]

remove xs ds      = if (length ds == 1) then ds
                    else ds \\ xs
```

Performance

- This program performs much better on very easy puzzles

```
solution ["-52439817", "8-9165432", "41-872596",
          "548-97321", "9315-4768", "26738-945",
          "795213-84", "1849562-3", "32674815-"]

["652439817","879165432","413872596"
,"548697321","931524768","267381945"
,"795213684","184956273","326748159"]
(0.01 secs, 513,832 bytes)
```

Performance

- This program performs much better on very easy puzzles

```
solution ["-52439817", "8-9165432", "41-872596",
          "548-97321", "9315-4768", "26738-945",
          "795213-84", "1849562-3", "32674815-"]

["652439817","879165432","413872596"
,"548697321","931524768","267381945"
,"795213684","184956273","326748159"]
(0.01 secs, 513,832 bytes)
```

- But this struggles on even puzzles with 39 entries

Performance

- This program performs much better on very easy puzzles

```
solution ["-52439817", "8-9165432", "41-872596",
          "548-97321", "9315-4768", "26738-945",
          "795213-84", "1849562-3", "32674815-"]

["652439817","879165432","413872596"
,"548697321","931524768","267381945"
,"795213684","184956273","326748159"]
(0.01 secs, 513,832 bytes)
```

- But this struggles on even puzzles with 39 entries
  - Aborted after running it for 6 hours on my laptop

Further improvements?

- **Improved strategy**

Further improvements?

- **Improved strategy**
  - Expand one cell at a time

Further improvements?

- **Improved strategy**
  - Expand one cell at a time
  - Interleave expansion and pruning

Further improvements?

- **Improved strategy**
  - Expand one cell at a time
  - Interleave expansion and pruning
- Instead of expanding all cells in the matrix of choices …

Further improvements?

- Improved strategy
  - Expand one cell at a time
  - Interleave expansion and pruning
- Instead of expanding all cells in the matrix of choices ...
- expand only one cell at a time.

Further improvements?

- **Improved strategy**
  - Expand one cell at a time
  - Interleave expansion and pruning
- Instead of expanding all cells in the matrix of choices ...
- expand only one cell at a time.
- A good choice is the smallest non-singleton cell

Further improvements?

- **Improved strategy**
  - Expand one cell at a time
  - Interleave expansion and pruning
- Instead of expanding all cells in the matrix of choices ...
- expand only one cell at a time.
- A good choice is the smallest non-singleton cell
- We now get a list of matrices

Further improvements?

- **Improved strategy**
  - Expand one cell at a time
  - Interleave expansion and pruning
- Instead of expanding all cells in the matrix of choices ...
- expand only one cell at a time.
- A good choice is the smallest non-singleton cell
- We now get a list of matrices
  - Each of these matrices contain singleton as well as non-singleton cells

Further improvements?

- **Improved strategy**
  - Expand one cell at a time
  - Interleave expansion and pruning
- Instead of expanding all cells in the matrix of choices …
- expand only one cell at a time.
- A good choice is the smallest non-singleton cell
- We now get a list of matrices
  - Each of these matrices contain singleton as well as non-singleton cells
- Prune each of these matrices

Further improvements?

- **Improved strategy**
  - Expand one cell at a time
  - Interleave expansion and pruning
- Instead of expanding all cells in the matrix of choices ...
- expand only one cell at a time.
- A good choice is the smallest non-singleton cell
- We now get a list of matrices
  - Each of these matrices contain singleton as well as non-singleton cells
- Prune each of these matrices
- And expand one cell in each incomplete matrix that results

Expanding one cell

- Expanding the smallest non-singleton cell:

```
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
```

Safe grids, complete grids

- A matrix of choices is safe if none of the singleton cells clash

```
safe :: Matrix Choices -> Bool
safe m          = all ok (rows m) &&
                  all ok (cols m) && all ok (boxs m)
  where
    ok row = nodups [d | [d] <- row]
```

Safe grids, complete grids

- A matrix of choices is safe if none of the singleton cells clash

```
safe :: Matrix Choices -> Bool
safe m          = all ok (rows m) &&
                  all ok (cols m) && all ok (boxs m)
   where
      ok row = nodups [d | [d] <- row]
```

- We can stop expanding if the matrix consists only of singleton entries

```
complete :: Matrix Choices -> Bool
complete              = all (all singleton)
   where singleton l = length l == 1
```

Searching for safe, complete grids

- Recall:

```
choices  = map (map choice)
choice d = if blank d then digits else [d]
```

Searching for safe, complete grids

- Recall:

```
choices  = map (map choice)
choice d = if blank d then digits else [d]
```

- To find all solutions, we search after creating a matrix of choices

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

Searching for safe, complete grids

- We alternate pruning and expanding a cell for incomplete grids

```
search :: Matrix Choices -> [Grid]
search m
    | not (safe m') = []
    | complete m'   = [map (map head) m']
    | otherwise     = concat (map search (expand1 m'))
      where m'      = prune m
```

Performance

- Works very well on easy inputs (39 cells filled in):

```
solution ["-5-43-81-", "-------3-", "-13--2---",
          "--8-9---1", "9-15-4-68", "-67---945",
          "795----84", "-8-956---", "32-748-59"]

["652439817","879165432","413872596"
,"548697321","931524768","267381945"
,"795213684","184956273","326748159"]
(0.02 secs, 3,129,904 bytes)
```

Performance

- Quite well on puzzles of higher difficulty (only 25 cells filled in):

```
solution ["--1----7-", "--7-18---", "------59-",
          "-2-------", "---35-1--", "------963",
          "-3--9---5", "4---63---", "59---7-48"]

["261935874","957418236","843276591"
,"329681457","674359182","185742963"
,"732894615","418563729","596127348"]
(0.02 secs, 9,082,360 bytes)
```

Performance

- Holds its own on against puzzles of very high difficulty (only 17 entries):

```
solution ["-12--5---", "---3---7-", "---------",
          "7--84----", "3-----9--", "------1--",
          "-9--12---", "6--5---8-", "---------"]

["812975364","946381572","537264819"
,"751849623","384126957","269753148"
,"498612735","623597481","175438296"]
(14.42 secs, 9,823,352,008 bytes)
```

Summary

- Incremental design of a non-trivial program

Summary

- Incremental design of a non-trivial program
- Reasonably simple logic, close to how a human would solve

Summary

- Incremental design of a non-trivial program
- Reasonably simple logic, close to how a human would solve
- Power of laziness – backtracking is easily programmed