A simple Sudoku solver in Haskell

Felix Döppers & Tobias Möllmann

Fachhochschule Wedel

14.01.2010

Inhaltsverzeichnis I

1. Einleitung
   - Was ist Sudoku?
   - Beispiel
2. Spezifikation
   - Datentypen
   - Mögliche Zellbelegungen
   - Expansion
   - Validierung
   - Ermittlung einzelner Zeilen, Spalten und Blöcke
   - Lösungsalgorithmus
3. Pruning
   - Hintergrund
   - Idee
   - Konkretisierung

Inhaltsverzeichnis II

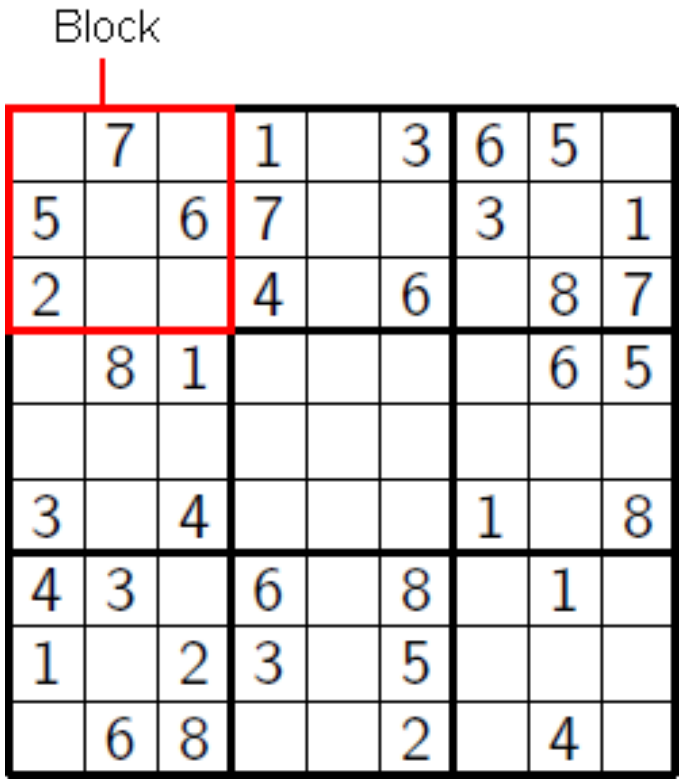
4 Single-cell Erweiterung
- Ansatz
- Implementierung

5 Fazit

Einleitung

Was ist Sudoku?

- Logikrätsel, welches aus einer 9x9-Matrix besteht
- Felder müssen mit den Ziffern 1-9 gefüllt werden, sodass jede Ziffer nur ein mal vorkommt in
  - jeder Spalte
  - jeder Zeile
  - jedem Block



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

Beispiel

Initialzustand:

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

Lösung:

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

Spezifikation

Datentypen

Definition einer Matrix

```
type Matrix a = [Row a]
type Row a = [a]
```

Spielfeld ist eine 9x9 Matrix mit Ziffern

```
type Grid = Matrix Digit
type Digit = Char
```

Mögliche Ziffern

```
digits = ['1'..'9']
blank = (=='0')
```

Mögliche Zellbelegungen

Menge möglicher Zellbelegungen

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

Ermittlung aller möglichen Zellbelegungen

```
choices :: Grid -> Matrix Choices
choices = map (map choice)
  where
  choice d = if blank d then digits else [d]
```

Expansion

expandiern

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

Kartesisches Produkt

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

Validierung I

Test, ob nur ein Vorkommen jeder Ziffer pro Zeile, Spalte und Block

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

Validierung II

Test, ob nur ein Vorkommen jeder Ziffer pro Zeile, Spalte und Block

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

Überprüfung, ob nur ein Vorkommen jeder Ziffer vorhanden

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

Ermittlung einzelner Zeilen, Spalten und Blöcke I

Zeilen

```
rows :: Matrix a -> Matrix a
rows = id
```

Spalten

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

Ermittlung einzelner Zeilen, Spalten und Blöcke II

Blöcke

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

group

```
group      :: [a] -> [[a]]
group []   = []
group xs   = take 3 xs : group (drop 3 xs)
```

ungroup

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

Lösungsalgorithmus

Zur Erinnerung

```
choices :: Grid -> Matrix Choices
expand  :: Matrix Choices -> [Grid]
valid   :: Grid -> Bool
```

Lösungsfunktion

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

Pruning

Hintergrund

- ein Sudokufeld besteht aus 81 Zellen
- durch Initalibelegung sind noch etwa 50 Zellen zu füllen
- für jede Zelle gibt es 9 verschiedene Belegungsmöglichkeiten
  - $\rightarrow$ insgesamt gibt es 9^{50} zu prüfende Möglichkeiten
  - $\rightarrow$ sehr ineffizient

Idee

Idee:

Lösche alle nicht mehr möglichen Choices!

Spezifikation

```
prune :: Matrix Choices -> Matrix Choices
```

Konkretisierung

Löschen von Choices

```
reduce :: Row Choices -> Row Choices
reduce xss = [xs \\ singles | xs <- xss]
  where singles = concat (filter single xss)
```

Funktionsdefinition

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

Single-cell Erweiterung

Ansatz

- selbst durch Einschränken der Bewegungsmöglichkeiten ist der Algorithmus noch zu langsam
- Ausweg: Single-cell Erweiterung:
  - Anbauen an einer einzigen Zelle anstatt an allen gleichzeitig
  - es wird die Zelle ausgewählt, die die geringste Belegungsmöglichkeiten hat
    - $\rightarrow$ schnelle Identifikation von Zellen mit keiner Belegungsmöglichkeit
    - $\rightarrow$ frühes Auffinden von unlösbaren Rätseln

Implementierung I

Schritt 1: Auswahl der besten Zeile

```
(rows1, row : rows2) = break (any smallest) rows
```

Schritt 2: Auswahl der Zelle mit geringsten Belegungsmöglichkeiten

```
(row1, cs : row2)     = break smallest row
```

mit:

```
smallest cs           = length cs == n
n                     = minimum (counts rows)
counts                = filter (/= 1) . (map length)
                        . concat
```

Implementierung II

expand1

```
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
```

Implementierung III

- Funktion nur sinnvoll für Matrizen mit wenigstens einer Zelle mit mehr als einer Belegungsmöglichkeit
- Matrizen mit mehreren gleichen Zahlen in einer Zeile, einer Spalte oder einem Block können verworfen werden

complete und safe

```
complete :: Matrix Choices -> Bool
complete =  all (all isSingleton)

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

ok       :: Row Choices -> Bool
ok       =  nodups . concat . filter isSingleton
```

Implementierung IV

Neue Lösungsfunktion

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

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
```

Fazit

- ein einfacher, intuitiver Algorithmus zum Lösen von Sudokurätseln ist entstanden
- kurzer Code im Vergleich zu anderen Lösungen
- Laufzeiteffizienz vergleichbar mit anderen Sudokulösern