

# Sudoku in Haskell

Sven Eric Panitz

8. Mai 2019

## Inhaltsverzeichnis

<b>1</b>	<b>Sudoku</b>	<b>1</b>
1.1	Lösungsstrategien . . . . .	2
<b>2</b>	<b>Haskell Implementierung</b>	<b>2</b>
2.1	Datentypen . . . . .	3
2.2	Lesen und Schreiben . . . . .	4
2.2.1	Einlesen eines Sudoku . . . . .	4
2.2.2	Darstellung für L <sup>A</sup> T <sub>E</sub> X . . . . .	4
2.3	Transformierte Darstellungen . . . . .	5
2.4	Test auf gelöste Sudoku . . . . .	6
2.5	Einschränken des Suchraums . . . . .	7
2.6	Lösen von Sudoku . . . . .	9
2.7	Beispiele . . . . .	10
<b>3</b>	<b>Lernzuwachs</b>	<b>11</b>

## 1 Sudoku

Sudoku (japanisch 数独 Sūdoku) sind eine Form Zahlenrätseln. In einer Tabelle mit 9 Zeilen und 9 Spalten, sind die Ziffern 1 bis 9 so einzutragen, dass keine Ziffer doppelt in einer Spalte oder Zeile auftritt. Hinzu gibt es 9 Unterquadrate von 3 Feldern Länge, in denen auch keine Ziffer mehrfach auftreten darf. Ein Sudokurätsel enthält ausgefüllte Felder. Aufgabe ist es das Sudoku komplett auszufüllen, ohne eine der Regeln zu verletzen. Ein Beispiel findet sich in Abbildung 1.

Nach [Arp18] wurden Sudoku 1979 von Howard Garns erfunden. Populär wurden sie zunächst in Japan und verbreiteten sich in den folgenden Jahrzehnten weltweit. Auch in Deutschland sind die Rätsel seit den 2000er Jahren sehr beliebt.

Das Prinzip ist leicht zu verstehen und bietet sich als Programmierübung gut an.

In Haskell gibt es viele Lösungen und auf [Has] gibt es eine kleine Übersicht. Besonders hervor zu haben ist der Artikel von Bird[Bir06], der auch in [Bir10] wieder in Druck erhältlich ist.

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

Abbildung 1: Ein Beispiel Sudoku, das als schwer gilt.

### 1.1 Lösungsstrategien

Ein reines Hauruckverfahren (*eng. brute force*), bei dem jedes Feld mit allen möglichen Ziffern ausgefüllt wird, und dann aus den so entstandenen ausgefüllten Sudoku die zu filtern, die die Regeln nicht verletzen, führt nur bei einfachen Sudoku zur Lösung. Das Sudoku aus Abbildung 1 führt dabei zu 7608096705012014379958272000000000000000000 zu prüfenden Sudoku. Nur sehr leichte Sudoku führen schon auf diesen Weg zu genau einen auszufüllenden gültigen Sudoku.

## 2 Haskell Implementierung

In diesem Aufgabenblatt werden wir einen Sudoku Lösungsalgorithmus implementieren.

### Haskell: Sudoku

```
> module Sudoku where
> import Data.Char
> import Data.List
```

Wir entwickeln in dieser Aufgabe eine einfache Lösung, die nicht auf eine Indizierung der Felder baut. Wir benötigen auch keine Datenstruktur, die zusätzliche Information während der Lösung verarbeitet. Die einzige Datenstruktur, die wir verwenden, sind die Standardlisten aus Haskell.

Damit gehen wir den Weg, den auch Bird in seiner Lösung gewählt hat. Anders als Bird werden wir aber nicht schrittweise über algebraische Vereinfachungen von einem Hausruckverfahren zu einer Lösung kommen, die auch komplexe Sudoku lösen kann, sondern direkt ein solches Lösungsverfahren anstreben.

## 2.1 Datentypen

Die 9x9 Tabelle eines Sudoku wird Liste von Listen dargestellt. Wir haben also eine Liste von 9 Zeilen, die 9 Elemente enthalten.

### Haskell: Sudoku

```
> type Sudoku a = [[a]]
```

Die Einträge der einzelnen Felder sollen alle noch denkbaren Ziffern sein, die in diesem Feld noch möglich sind. Auch hierfür verwenden wir eine Liste. Ist diese Liste für ein Feld leer, dann gibt es für das Sudoku keine Lösung. Hat diese Liste genau ein Element, so steht der Eintrag für dieses Feld fest. Ansonsten gibt es noch eine Liste möglicher Optionen für das entsprechende Feld.

### Haskell: Sudoku

```
> type Options = [Int]
> type SudokuPuzzle = Sudoku Options
```

In keiner Zeile darf eine leere Liste stehen, denn dann ist für uns ein Sudoku unlösbar.

### Haskell: Sudoku

```
> unsovable :: SudokuPuzzle -> Bool
> unsovable = or.map (elem [])
```

## 2.2 Lesen und Schreiben

### 2.2.1 Einlesen eines Sudoku

Um einfach Sudokus zu lesen, benötigen wir eine Funktion, die eine Stringdarstellung liest und in unseren Datentyp `Sudoku` umwandelt. Der String soll dabei aus neun Zeilen bestehen, die genau 9 Zeichen lang sind. Die Zeichen sind entweder Ziffern, das sind die vorausgefüllten Felder des Sudoku, oder ein beliebiges sonstiges Symbol. Dieses sind dann die freien Felder des Sudoku.

Das Sudoku aus Abbildung 1 soll also aus folgenden String eingelesen werden können.

Haskell: Sudoku

```
> sc4 =  
> "....6..8.\n\  
> \.2.....\n\  
> \..1.....\n\  
> \.7....1.2\n\  
> \5...3....\n\  
> \.....4..\n\  
> \..42.1...\n\  
> \3..7..6..\n\  
> \.....5."
```

#### Aufgabe 1

Schreiben Sie eine Funktion `readSudoku`, die ein Sudoku aus einer Stringdarstellung einliest. Vorausgefüllte Felder sollen dabei zu einelementigen Listen werden, nicht ausgefüllte Felder sollen zur Liste `[1..9]` werden.

Haskell: Sudoku

```
> readSudoku :: String -> SudokuPuzzle  
> readSudoku xs = []
```

### 2.2.2 Darstellung für $\text{\LaTeX}$

Um Sudokus bequem wieder in der gewohnten Druckdarstellung zu bekommen, können wir uns des Satzprogramms  $\text{\LaTeX}$  bedienen. Für  $\text{\LaTeX}$  gibt es ein eigenes *Package* zum Satz von Sudokus[Abr06].

Der Quelltext für ein Sudoku ist dabei denkbar einfach und intuitiv. Folgendes Beispiel ist der  $\text{\LaTeX}$ -Quelltext für das Sudoku aus Abbildung 1.

```
\begin{sudoku}
```

```

| | | | 6| | 8| |.
| 2| | | | | |.
| | 1| | | | |.
| 7| | | | 1| 2|.
| 5| | | 3| | |.
| | | | | 4| |.
| | 4| 2| 1| | |.
| 3| | 7| | 6| |.
| | | | | | 5| |.
\end{sudoku}

```

## Aufgabe 2

Schreiben Sie die Funktion `toLatex`, die ein Sudoku für das  $\text{\LaTeX}$ -Paket als String darstellt. Felder mit mehr als einer Option bleiben dabei leer. Nur Zellen, in denen nur eine Ziffer in der Liste steht, werden als ausgefüllt dargestellt.

### Haskell: Sudoku

```

> toLatex :: SudokuPuzzle -> String
> toLatex xss = ""

```

## 2.3 Transformierte Darstellungen

In unserer Datenstruktur bevorzugen wir Zeilen. Die Elemente einer Zeile stehen direkt in einer Liste. Diese kann einfach auf doppelte Einträge getestet werden. Die Elemente einer Spalte hingegen sind schwer zu greifen. Um an die dritte Spalte zu kommen, müssten wir folgenden Ausdruck auf ein Sudoku anwenden: `map (!!2)`. Der Zugriff über den Index auf eine Liste ist für die Standardlisten eine ineffiziente Funktion, die jeweils die Liste von vorne durchlaufen muss.

Deshalb werden wir Funktionen vorsehen, die eine Darstellung des Sudokus als Listen der Spalten transformiert. Glücklicherweise brauchen wir diese Funktion gar nicht zu schreiben, weil diese sich bereits im Standard-API unter den Namen `transpose` befindet.

Somit lässt sich die Liste der Spalten über die Funktion `transpose` erzeugen. Die Reihen eines Sudokus finden sich in unserer Datenstruktur direkt.

### Haskell: Sudoku

```

> columns, rows :: Sudoku a -> Sudoku a
> rows      = id
> columns   = transpose

```

Die Darstellung als Liste der Zeilen oder als Liste der Spalten ist für die Lösung eines Sudokus unerheblich. Die 3x3 Quadrate finden sich dabei ebenso zusammengesetzt. Wir können also beliebig oft auf ein Sudoku die Funktion `transpose` während der Lösungssuche anwenden.

Insbesondere gilt auch:

$$id = columns \circ columns$$

Spannender sind die 3x3 Quadrate. Um schnell zu checken, ob eine Bedingung in einem dieser Quadrate verletzt ist, ist eine Darstellung wünschenswert, in der die 9 Felder eines solchen Quadrates in einer Liste gebündelt sind. Wir suchen also eine Liste der Quadrate eines Sudokus.

### Aufgabe 3

Implementieren Sie die Funktion `boxes`, die für einen Sudoku eine Liste aller Unterquadrate erzeugt. Die Quadrate sollen dabei erst spaltenweise von oben nach unten und dann von links nach rechts im Ergebnis stehen.

Haskell: Sudoku

```
> boxes :: Sudoku a -> Sudoku a
> boxes xss = xss
```

Zeigen Sie, das gilt:

$$id = boxes \circ boxes$$

Anders als bei der Funktion `columns` stellt das Sudoku, dass die Funktion `boxes` erzeugt ein anderes Sudoku dar. Lösungen von `(boxes xss)` sind nicht unbedingt auch Lösung von `xss`.

## 2.4 Test auf gelöste Sudoku

Zum Testen, ob neun Felder eines fertig ausgefülltes Sudoku korrekt belegt sind, dient die folgende kleine Funktion.

Haskell: Sudoku

```
> correct :: Eq a => [a] -> Bool
> correct xs = 9 == length (nub xs)
```

Ein fertig ausgefülltes Sudoku ist eine Lösung, wenn jede Zeile, jede Spalte und jedes 3x3 Quadrat korrekt belegt ist.

### Haskell: Sudoku

```
> solved xss = okay (rows xss) && okay (columns xss) && okay (boxes xss)
> where okay = and.(map correct)
```

Die Funktionen dieses Abschnitts gehen davon aus, dass jedes Feld durch eine einelementige Liste belegt ist.

## 2.5 Einschränken des Suchraums

Die Sudoku, die wir aus einem String einlesen, haben noch keine Restriktionen betrachtet. Für jedes Feld, das nicht ausgefüllt ist, gehen wir davon aus, dass dort jeder der 9 Ziffern stehen kann. Wenn wir eine einzelne Zeile, Spalte oder ein Quadrat betrachten, können wir aus den Optionen jeweils die Möglichkeiten streichen, die in einem der anderen 8 Felder bereits fest gesetzt sind. Damit lässt sich die Anzahl der auszuprobierenden Möglichkeiten für viele Sudoku schon beträchtlich einschränken.

### Aufgabe 4

Schreiben Sie die Funktion `limitLine`, die aus einer Reihe (bzw. Spalte oder Box) bei allen Optionen für Feldern die Ziffern streicht, die bereits in anderen Feldern fest vorgegeben sind.

#### Haskell: Sudoku

```
> limitLine :: [Options] -> [Options]
> limitLine lls = lss
```

Damit können wir für ein Sudoku alle direkt sichtbaren Einschränkungen berücksichtigen. Wir berücksichtigen nacheinander die Zeilen, die Spalten und dann die 3x3 Quadrate. Da die Funktion `boxes` nicht ein lösungsäquivalentes Sudoku erzeugt, ist sie schließlich wieder zum Herstellen der ursprünglichen Darstellung anzuwenden.

### Haskell: Sudoku

```
> limitation :: SudokuPuzzle -> SudokuPuzzle
> limitation
> = boxes
>   .map limitLine.boxes
>   .map limitLine.columns
>   .map limitLine.rows
```

Wie viele Versionen verbleiben uns dann potentiell zu probieren? Hierzu können wir das Produkt der Anzahl der Optionen der einzelnen Felder berechnen.

### Haskell: Sudoku

```
> versions :: SudokuPuzzle -> Integer
> versions = (product.map (toInteger.length).concat)
```

Die Funktion `limitation` betrachtet nacheinander einzelne Zeilen, Spalten und Quadrate und schränkt bezüglich dieser den Suchraum ein. Dieses kann jeweils dazu führen, dass ein Feld keine Optionen mehr hat, sondern damit schon fest gesetzt ist. Dieses führt dazu, dass damit wieder andere Einschränkungen festgelegt werden. Es lohnt sich also, mehrfach hintereinander die Funktion `limitation` anzuwenden, bis die Anzahl der Möglichkeiten sich nicht mehr verringert.

Hierzu dient die folgende Funktion:

### Haskell: Sudoku

```
> limitations lls
> |versions lls' == versions lls = lls
> |otherwise = limitations lls'
>   where
>     lls' = limitation lls
```

Damit lassen sich einfache Sudoku schon komplett lösen. Hierzu betrachte man das folgende Sudoku:

### Haskell: Sudoku

```
> sc3 =
>   ".26...81.\n\
>   \3..7.8..6\n\
>   \4...5...7\n\
>   \.5.1.7.9.\n\
>   \..39.51..\n\
>   \.4.3.2.5.\n\
>   \1...3...2\n\
>   \5..2.4..9\n\
>   \.38...46."
```

In diesem Sudoku sind 47 Felder nicht vorbelegt. Naiv kann also jedes dieser 47 Felder durch 9 Ziffern belegt werden.

Insgesamt also  $9^{47} = 706965049015104706497203195837614914543357369$  Versionen.

Betrachten wir nacheinander die Einschränkungen ergibt sich:

```
*Sudoku> versions$limitation$readSudoku sc3
138663194171277312
*Sudoku> versions$limitation$limitation$readSudoku sc3
3439853568
```



```

*Sudoku> versions$limitation$limitation$limitation$readSudoku sc3
32
*Sudoku> versions$limitation$limitation$limitation$limitation$readSudoku sc3
1
*Sudoku>

```

Dieses Sudoku lässt sich also durch wiederholtes Propagieren der Einschränkungen mit der Funktion `limitations` lösen.

## 2.6 Lösen von Sudoku

Nur die einfachsten Sudoku lassen sich allein durch Propagieren der Einschränkungen direkt lösen. Komplexere Sudoku verlangen nach wie vor eine Suche, in der systematisch nach und nach die Möglichkeiten ausprobiert werden.

Unsere Strategie wird sein, zeilenweise nach dem ersten nicht festgelegten Feld zu suchen. Dann wird für alle die Optionen, die dieses Feld noch hat, ein Sudoku erstellt.

### Aufgabe 5

Schreiben Sie die Funktion `firstFillings` die alle Sudoku erzeugt, die entstehen, wenn eine der Optionen des ersten Feldes, das noch nicht festgelegt wird, gewählt wird.

Haskell: Sudoku

```

> firstFillings :: SudokuPuzzle -> [SudokuPuzzle]
> firstFillings xss = []

```

Die Funktion `firstFillings` ist der Kern unserer Lösungsstrategie.

Um ein Sudoku zu lösen, propagiere zunächst alle Einschränkungen, die direkt zu erkennen sind. Wenn es dann keine Möglichkeit gibt, es auszufüllen, war es nicht lösbar. Wenn es nur eine Möglichkeit gibt, es auszufüllen, so ist das Sudoku gelöst.

Ansonsten: betrachte alle Möglichkeiten des ersten nicht festgelegten Feldes. Versuche diese jeweils rekursiv zu lösen. Wir erhalten eine Liste von Listen. Die meisten dieser Listen werden leer sein, weil dort keine Lösung mehr zu finden war.

### Haskell: Sudoku

```
> solve xs
> |vs==0 = []
> |vs==1 && solved lxs = [lxs]
> |otherwise = concat$map solve$firstFillings lxs
> where
>   lxs = limitations xs
>   vs = versions lxs
```

## 2.7 Beispiele

In diesem Papier waren schon zwei Extrembeispiele für Sudoku gegeben. `sc4`, das extrem schwer ist und auf [Has] als *nefarious* bezeichnet wird und `sc3`, das extrem leicht ist, da es allein durch Propagieren der Einschränkungen gelöst werden kann.

Hier noch zwei Sudoku, die zwischen den Extremen liegen.

### Haskell: Sudoku

```
> sc2 =
>   "8156....4\n\
>   \6...75.8.\n\
>   \....9....\n\
>   \9...417..\n\
>   \.4.....2.\n\
>   \..623...8\n\
>   \....5....\n\
>   \.5.91...6\n\
>   \1....7895";
```

### Haskell: Sudoku

```
> sc5 =
>   ".5..6...1\n\
>   \..48...7.\n\
>   \8.....52\n\
>   \2...57.3.\n\
>   \.....\n\
>   \.3.69...5\n\
>   \79.....8\n\
>   \.1...65..\n\
>   \5...3..6."
```

### Aufgabe 6

Testen Sie jetzt Ihre Implementierung, in dem Sie sie in einer kompilierten Version laufen lassen. Nennen Sie das Modul hierzu um in `Main` und nennen Sie die folgende Funktion um in `main`. Dann kompilieren Sie das Programm mit der Anweisung `ghc -O2 Sudoku.lhs`. Lassen Sie das Programm laufen und beobachten Sie Speicherverhalten und Laufzeit.

#### Haskell: Sudoku

```
> moin = do
> print $solve $readSudoku sc2
> print $solve $readSudoku sc3
> print $solve $readSudoku sc4
> print $solve $readSudoku sc5
```

Machen Sie weitere Tests mit unterschiedlichen Sudoku.

### Aufgabe 7

Vergleichen Sie unsere Lösung mit den unterschiedlichen Lösungen auf [Has] sowohl in Hinblick auf die algorithmische Lösung als auch auf das Laufzeit und Speicherverhalten.

## 3 Lernzuwachs

Folgende Lerninhalte konnten mit dieser Aufgabe gut erarbeitet werden:

- Arbeiten mit den Standardlisten aus dem Haskell Prelude.
- Erste Beispiele für Gesetze auf Funktionen.
- Einsatz von Typsynonymen.
- Algorithmisches vereinfachen von komplexen Suchproblemen.

## Literatur

- [Abr06] Abraham, Paul, “The sudoku package,” <http://ftp.uni-erlangen.de/ctan/macros/latex/contrib/sudoku/sudoku.pdf>, 2006, [Online; accessed 8-May-2019].
- [Arp18] R. Arp, *1001 Ideas that Changed the Way We Think*, ser. 1001. Octopus, 2018. [Online]. Available: <https://books.google.de/books?id=HGNADwAAQBAJ>

- [Bir06] R. S. Bird, “Functional pearl: A program to solve sudoku,” *J. Funct. Program.*, vol. 16, no. 6, pp. 671–679, 2006.
- [Bir10] R. Bird, *Pearls of Functional Algorithm Design*, 1st ed. New York, NY, USA: Cambridge University Press, 2010.
- [Has] Haskell.org, “Sudoku. Here are a few Sudoku solvers coded up in Haskell...” [accessed 07-May-2019]. [Online]. Available: <https://wiki.haskell.org/Sudoku>