Sudoku

Projekte aufsetzen: cabal'ismus
Programmkonstruktion via *Equational Reasoning!*

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Jan 19^{th}, 2023

Echtwelt-Haskell: Projektdefinition

- `.cabal` definieren Haskell-Projekte
- beschreiben Abhängigkeiten von anderen Libraries
- definieren Executables, Libraries, Automatische Tests
- `cabal` (das Programm) berechnet für die Abhängigkeiten die Versionsgrenzen und konstruiert einen Bauplan. Dieser kann abgespeichert werden.
- Automatische Tests können auch in github bei jedem commit starten
- `hackage.haskell.org` ist ein Repository mit vielen Libraries die alle unter freien Lizenzen stehen

```
cabal-version:   3.0
name:            Sudoku
version:         0.1.0.0
synopsis:        A simple sudoku solver following Pearls of
license:         BSD-3-Clause
license-file:    LICENSE
author:          Christian Hoener zu Siederdissen
maintainer:      christian.hoener.zu.siederdissen@uni-jena.de

library
    exposed-modules:  Sudoku
    build-depends:    base ^>=4.15.1.0
                    ,   deepseq
    hs-source-dirs:   lib
    default-language: Haskell2010
```

Sudoku

In jeder Zeile, Spalte, 3×3 Submatrix komme jede Ziffer aus $\{1, \ldots, 9\}$ nur einmal vor.

```
 1       _ 5 _   _ 6 _   _ _ 1
 2       _ _ 4   8 _ _   _ 7 _
 3       8 _ _   _ _ _   _ 5 2
 4
 5
 6       2 _ _   _ 5 7   _ 3 _
 7       _ _ _   _ _ _   _ _ _
 8       _ 3 _   6 9 _   _ _ 5
 9
10
11       7 9 _   _ _ _   _ _ 8
12       _ 1 _   _ _ 6   5 _ _
13       5 _ _   _ 3 _   _ 6 _
```

Grundlegende Definitionen

```
 1  -- | Matrix definiert als Liste von Zeilen
 2  type Matrix a = [Row a]
 3  -- | Zeile definiert als Liste Zellen vom Typ @a@
 4  type Row a = [a]
 5  -- | Das Sudoku Grid, als Matrix von Buchstaben
 6  type Grid = Matrix Digit
 7  -- | Repraesentiere Digits via Chars
 8  type Digit = Char
 9  -- | Zelle ist noch nicht fixiert: Wahlmoeglichkeiten
10  type Choices = [Digit]
11
12  -- | Legale Eintraege
13  digits :: [Char]
14  digits = ['1' .. '9']
15
16  -- | Eintrag ist noch "leer"
17  blank :: Char -> Bool
18  blank = (=='0')
```

Sudoku via Definition

- Idee: wir definieren, was eine Sudoku-Lösung ist
- durch *equational Reasoning* finden wir semantisch gleiche Definitionen
- diese neuen Definitionen rechnen allerdings (beweisbar!) schneller
- dadurch planen und konstruieren wir einen korrekten, effizienten Sudoku-Solver

Prinzipiell können wir `Puzzle.hs` nutzen, das ist eine gute Übung für zu Hause.

Eine primitive Lösung für Sudoku

- Gegeben ein `Grid`
- Nutze `choices` um alle "Löcher"mit allen Kandidaten $\{1, \ldots, 9\}$ zu füllen
- Nehme dann dieses Metasudoku und generiere mittels *kartesischem Produkt* alle möglichen Grids
- Teste jedes Grid darauf ob es eine "legale"Lösung darstellt
- . . . Profit!
- (alle folgenden Funktionen sollten so “einfach” sein, das sie sofort als richtig erkannt werden können)

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

Jetzt noch schnell die fehlenden Funktionen schreiben . . .

choices

- `choices` bearbeitet jede Zelle, nutzt `choice`
- `choice` betrachtet eine Zelle mit Wert d, ist dieser Wert nicht gesetzt ("blank')), dann sind alle 9 Möglichkeiten in der Zelle, ansonsten der (vorher) fixierte Wert

```
1  -- | Gegeben ein grid, erstelle eine Matrix
2  -- in der jede Zelle mit den moeglichen 'Choices', also
3  -- einer Liste der moeglichen 'Digit's gefuellt ist.
4  choices :: Grid -> Matrix Choices
5  choices = map (map choice)
6
7  -- | Choice testet ob eine Zelle noch frei ist und gibt
8  -- dann alle Kandidaten zurueck, ansonsten wird
9  -- der schon gesetzte Wert zurueck gegeben.
10 choice :: Char -> [Char]
11 choice d = if blank d then digits else [d]
```

expand: Kartesisches Produkt

expand generiert alle möglichen Lösungen, was passiert hier?

(Klar / Whiteboard)?

$$\begin{matrix} [1,2] & [3] \\ [4] & [5,6] \end{matrix}$$

```
1  -- | Erstelle aus der Matrix der Choices die
2  -- Liste aller Kandidatengrids.
3  expand :: Matrix Choices -> [Grid]
4  expand = cartProd . map cartProd
5
6  -- | Das kartesische Produkt von @n@ Listen
7  cartProd :: [[a]] -> [[a]]
8  cartProd [] = [[]]
9  cartProd (xs:xss) = [x:ys | x <- xs, ys <- cartProd xss]
```

Korrektheitstest

```
 1  -- | Teste jedes Grid darauf, ob es ein Sudoku ist.
 2  valid :: Grid -> Bool
 3  valid g =  all nodups (rows g) && all nodups (cols g)
 4          && all nodups (boxs g)
 5
 6  -- | Teste rekursiv ob Duplikate vorkommen,
 7  nodups :: Eq a => [a] -> Bool
 8  nodups [] = True
 9  nodups (x:xs) = notElem x xs && nodups xs
10
11  -- | selektiert alle Zeilen
12  rows :: Matrix a -> Matrix a
13  rows = id
14
15  -- | Selektiert alle Spalten; siehe auch 'transpose'
16  cols :: Matrix a -> Matrix a
17  cols [xs] = [[x] | x <- xs]
18  cols (xs:xss) = zipWith (:) xs (cols xss)
```

3 × 3 Boxen

```
 1  -- | Selektiert alle 3x3 Quadrate
 2  boxs :: Matrix a -> Matrix a
 3  boxs = map ungroup . ungroup
 4       . map cols . group . map group
 5
 6  -- | Neuorientierung in 3er-Gruppen:
 7  -- @group [1,2,3,4,5,6] == [[1,2,3],[4,5,6]]@
 8  group :: [a] -> [[a]]
 9  group [] = []
10  group xs = take 3 xs : group (drop 3 xs)
11
12  -- | Hebt eine Gruppierung auf
13  ungroup :: [[a]] -> [a]
14  ungroup = concat
```

Fertig

- Prinzipiell sind wir fertig
- der Algorithmus is vglw. leicht verständlich
- und Korrektheit könnte gezeigt werden
- die Performance ist allerdings unterirdisch . . .

1. Optimierung: *Pruning*

```
 1  -- | Entfernt illegale Auswahlen
 2  prune :: Matrix Choices -> Matrix Choices
 3  prune = pruneBy boxs . pruneBy cols . pruneBy rows
 4
 5  -- | Intern wird nur nach "rows" getestet, daher muss in
 6  -- Rows umgewandelt und zurueck verwandelt werden.
 7  pruneBy :: ([Row Choices] -> [Row Choices])
 8          -> [Row Choices] -> [Row Choices]
 9  pruneBy f = f . map pruneRow . f
10
11  pruneRow :: Row Choices -> Row Choices
12  pruneRow row = map (remove fixed) row
13    where fixed = [d | [d] <- row]
14
15  -- | @xs@ entfernen, @ds@ die Kandidaten
16  remove xs ds = if singleton ds then ds else ds \\ xs
17
18  solveP :: Grid -> [Grid]
19  solveP = filter valid . expand . prune . choices
```

Expandiere Eine Zelle

```
 1  expand1 :: Matrix Choices -> [Matrix Choices]
 2  expand1 rows = [rows1 ++ [row1 ++ [c] : row2] ++ rows2
 3                  | c <- cs]
 4    where
 5      -- @row@ enthaelt irgendwo "n" Choices
 6      (rows1,row:rows2) = break (any smallest) rows
 7      -- @cs@ sind die erste Zelle bei der das so ist
 8      (row1,cs:row2) = break smallest row
 9      smallest cs = length cs == n
10      -- Min. ueber alle rows
11      n = minimum (counts rows)
12
13  counts :: [[Choices]] -> [Int]
14  counts = filter (/=1) . map length . concat
```

Rekursiv Lsg finden

```
1  -- | Gibt es noch Expaniermoeglichkeiten?
2  complete :: [[[a]]] -> Bool
3  complete = all (all singleton)
4
5  -- | Ist dieser Kandidat ok?
6  safe m = all ok (rows m) && all ok (cols m) && all ok (boxs
7    where ok row = nodups [ d | [d] <- row ]
8
9  search :: Matrix Choices -> [Matrix Digit]
10 search m
11   | not (safe m) = []
12   | complete m' = [map (map head) m']
13   | otherwise = concatMap search (expand1 m')
14   where m' = prune m
15
16 solve0 :: Grid -> [Grid]
17 solve0 = search . choices
```

filtered pruning

Pruning entfernt nur "Duplikatslösungen"

```
1  filter nodups . cartProd = filter nodups . cp . pruneRow
```

Mathematisch: Eine *Involution f* ist eine Funktion die ihr eigenes Inverses ist:

$$f(f(x)) = x$$

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

Und:

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

Zu zeigen

```
1  filter valid . expand
2  = filter (all nodups . boxs) .
3    filter (all nodups . cols) .
4    filter (all nodups . rows) . expand
5  = filter valid . expand . prune
```

Beweis für Boxes

```
 1  filter (all nodups . boxs) . expand
 2  = map boxs . filter (all nodups) . map boxs . expand
 3  = map boxs . filter (all nodups) . expand . boxs
 4  = map boxs . filter (all nodups) . cp . map cp . boxs
 5  = map boxs . cp . map (filter nodups . cp) . boxs
 6  = map boxs . cp . map (filter nodups . cp . pruneRow)
 7    . boxs
 8  = map boxs . filter (all nodups) . cp . map cp
 9    . map pruneRow . boxs
10  = map boxs . filter (all nodups) . expand
11    . map pruneRow . boxs
12  = filter (all nodups . boxs) . map boxs . expand
13    . map pruneRow . boxs
14  = filter (all nodups . boxs) . expand . boxs
15    . map pruneRow . boxs
```