Puzzle-Löser in Haskell

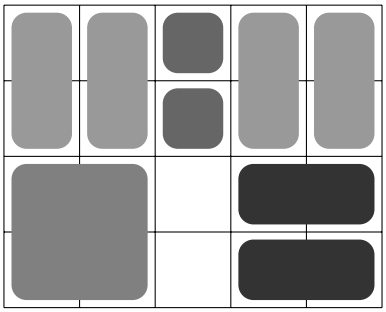
Joachim Breitner

8.5.2006

Folgendes Puzzle wurde mir kürzlich gestellt:

1 Aufgabenstellung

Gegeben ist ein Spielfeld mit $5x4$ Feldern, auf dem Spielsteine wie folgt angeordnet sind:



Die Felder in der Mitte unten sind leer. Diese Spielsteine sollen jetzt so verschoben werden, dass das Viereck unten links steht.

2 Lösung in Haskell

Da ich „von Hand“ nicht drauf kam, wie das Problem zu lösen ist, erstellte ich folgendes Haskellprogramm. Da dies auch für andere, vor allem für Besucher meines Informatik-II-Tutoriums, interessant sein könnte, dokumentiere und veröffentliche ich es hier.

Ich beginne mit dem Import einiger weniger Library-Funktionen. Dabei verstecke ich die „`lines`“-Funktion, da ich selbe eine solche habe.

```
0  import qualified Prelude(lines)
1  import Prelude hiding(lines)
2  import Data.List hiding (lines)
3  import System.IO
```

In meinem Programm schiebe ich viel Puzzlezustände hin und her, also definiere ich dafür einen neuen Typ, `HWState`. Der Typ `Coord` ist nur da, um etwas Schreibarbeit zu sparen. Der Typ ist ein Record mit Feldern für die Coordinaten der Spielsteine. Da es von senkrechten, waagrechten und kleinen Steinen mehrere gibt, verwende ich hier eine Liste. Ich muss später selbst drauf achten, dass die richtige Anzahl gespeichert ist. Auch werde ich die Coordinaten stets sortiert einspeichern, darauf verlässt sich der folgende Code auch.

```
4  type Coord = (Int, Int)
5  data HWState = HWState {
6          square::Coord,
7          bars::[Coord],
8          lines::[Coord],
9          dots::[Coord]
10 } deriving Eq — Damit schenkt mir Haskell ohne Mehraufwand die (==) und (/=)-Operatoren
```

Ich werde viel mit Coordinaten rumschieben, also dazu ein paar Funktionen:

```
11 rueber    (x,y) = (x+1,y)
12 runter    (x,y) = (x,y+1)
13 ruerunter (x,y) = (x+1,y+1)
14 hoch      (x,y) = (x,y-1)
15 links     (x,y) = (x-1,y)
```

Die Funktion `doeach` wendet die Funktion auf jeweils eines der Elemnete von `f` an und gibt eine Liste all dieser Möglichkeiten zurück.

```
16 doeach f []     = []
17 doeach f (x:xs) = ((f x):xs) : (map (x:) (doeach f xs))
```

Diese Funktion wird für die Funktion `bewegeach` gebraucht. Analog zu `beweg`, welche eine Koordinate in alle richtungen macht, macht `bewegeach` das für alle Koordinaten in der übergebenen Liste. Das ganze brauch ich für später, wenn ich die möglichen Züge berechnen will.

```
18 beweg c = [rueber c, runter c, hoch c, links c]
19 bewegeach l = (doeach runter l ++ doeach rueber l ++ doeach hoch l ++ doeach links l)
```

Ich will überprüfen können, ob ein Spielfeld gültig ist, sich also keine Spielsteine überlappen oder herausragen. Dazu habe ich für jede Spielsteinart eine Funktion, die die verwendeten Felder zurückgibt, sowie eine, die diese dann noch zusammenführt:

```
20 usedBySquare state  = [             square state,
21                        (rueber    $ square state),
22                        (runter    $ square state),
23                        (ruerunter $ square state) ]
24
25 usedByLines state  = lines state ++ map rueber (lines state)
26 usedByBars  state  = bars  state ++ map runter (bars  state)
27 usedByDots  state  = dots  state
28
29 used state  = usedBySquare state  ++ usedByLines state ++ usedByBars  state ++ usedByDots state
```

Gültig ist ein Feld natürlich zum einen, wenn nichts herausragt. Das überprüfen erstmal für jede Spielsteinart die folgenden Funktionen:

```
30 validSquare (HWState s _ _ _) =     (\(x,y) → 0 ≤ x && x ≤ 3 && 0 ≤ y && y ≤ 2) s
31 validBars   (HWState _ b _ _) = all (\(x,y) → 0 ≤ x && x ≤ 4 && 0 ≤ y && y ≤ 2) b
32 validLines  (HWState _ _ l _) = all (\(x,y) → 0 ≤ x && x ≤ 3 && 0 ≤ y && y ≤ 3) l
33 validDots   (HWState _ _ _ d) = all (\(x,y) → 0 ≤ x && x ≤ 4 && 0 ≤ y && y ≤ 3) d
```

Gültig letztendlich ist ein Feld genau dann, wenn alle Steine im Feld liegen und wenn keine Felder doppelt belegt sind. Letzeres erkenen ich daran, ob die Liste der belegten Felder noch die gleiche ist, wenn ich Doubletten mit **nub** entferne.

```
34 valid state = validSquare state &&
35               validLines  state &&
36               validBars   state &&
37               validDots   state &&
38               nub used' == used'
39               where used' = used state
```

Gelöst ist das Puzzle, wenn in einem Zustand das Viereck unten rechts ist:

```
40 solved state = (square state ) == (3,2)
```

Für die Pseudo-Graphische Ausgabe brauche ich Zeichenfunktionen. Die `pixs...`-Funktionen geben mir eine Liste von Tupeln zurück, die zum einen die Koordinate und zum anderen die Zeichen enthalten.

```
41 pixsSquare state = [ (            square state, "##"),
42                      (rueber    $ square state, "##"),
43                      (runter    $ square state, "##"),
44                      (ruerunter $ square state, "##") ]
45 pixsLines state = concatMap (\c→ [ (c,"≤"), (rueber c, "⇒") ] )   (lines state)
46 pixsBars  state = concatMap (\c→ [ (c,"/\\"), (runter c, "\\/") ] ) (bars  state)
47 pixsDots  state = concatMap (\c→ [ (c,"()")                    ] )   (dots  state)
```

`draw` sorgt dafür, das auch nicht belegte Pixel – also solche für die **lookup Nothing** zurück gibt – ihren Raum einnehmen.

```
48 draw Nothing  = "␣␣"
49 draw (Just c) = c
```

Nun kann ich Zeichnen. Dazu mach ich mein Spielfeldtyp doch gleich eine Instanz der Klasse **Show**. Ich berechne erst in `pixs` alle Pixel des Zustandes und suche dann der Reihe nach mit **lookup** danach. Wenn ich nichts finde, sort `draw` dafür, dass das kein Problem ist.

```
50 instance Show HWState where
51         show state = "\n" ++ concat [
52                         concat [draw $ lookup (x,y) pixs | x←[0..4] ] ++ "\n" | y ← [0..3] ]
53                       where pixs = pixsSquare state ++ pixsBars state ++
54                                    pixsLines state ++ pixsDots state
```

Die mehrzeilige Ausgabe ist etwas interessanter. Der Paramter `n` zählt die ausgegebenen Felder hoch. Die innere Zeile (**unlines**...) nimmt die ersten 6 auszugebenden Felder, zeichnet diese mit dem **show** von oben und spaltet diesen String in seine Zeilen auf. Wir haben jetzt eine Liste von einer Liste von Zeilen. Mit **transpose** sortieren wir diese um, so was wir die Zeilen jetzt nur noch mit **unwords** und **unlines** zusammensetzen müssen.

```
55 show' _ [] = ""
56 show' n l  = let (anf, end) = splitAt 6 l
57              in unlines ["Schritt␣"++(show n)++"-"++( show $ n + (length anf) - 1)++":",
58                          unlines $ map (unwords) $  transpose $ map Prelude.lines $ map show anf,
59                          show' (n + (length anf)) end]
```

Zurück zum Puzzle. Ein Zug bewegt genau ein Stein. Also für jede Steinsart eine Bewegungs-Funktion:

```
60 moveSquare (HWState s b l d) = map (\s → HWState s b l d) (beweg s)
61 moveBars   (HWState s b l d) = map (\b → HWState s (sort b) l d) (bewegeach b)
62 moveLines  (HWState s b l d) = map (\l → HWState s b (sort l) d) (bewegeach l)
63 moveDots   (HWState s b l d) = map (\d → HWState s b l (sort d)) (bewegeach d)
```

Und nun eine fürs ganze Feld. Da wir hier nun viele unsinnge Bewegungen drin haben, schmeißen wir die mit **filter** gleich wieder raus.

```
64 moves state = filter valid $
65               concat [
66               moveSquare state,
67               moveLines  state,
68               moveBars   state,
69               moveDots   state ]
```

Nicht zu vergessen: Die Startkonfiguration:

```
70 start = HWState (0,2)
71                 (sort [ (0,0),  (1,0),  (3,0), (4,0)])
72                 (sort [ (3,2),  (3,3)])
73                 (sort [ (2,0),  (2,1)])
```

Die folgende Funktion implementiert Breitensuche im Graphen, und terminiert wenn es eine Lösung gibt. Sie ist voll parametrisiert, benutzt also nichts vom Code weiter oben, und kann auch für viele andere Probleme verwendet werden. Dabei verrichtet die rekursive `seen'`-Funktion die Hauptarbeit: Der erste Parameter ist eine Liste von momentan untersuchten Wegen, also Listen von „benachbarten“ Zuständen mit dem Startzustand am Ende. Der zweite Parameter ist zur Optimierung und ist eine Liste aller bereits passierten Zustände.

Falls einer der übergebenen Wege in einem Zielzustand endet, findet **find** diesen und wir sind fertig. Ansonsten folgen wir mit **next'** jeder Möglichkeit, noch einen Zug zu machen, und erhalten eine neue Liste von Wegen. Diese verkleinern wir jetzt, in dem wir alle Wege rausschmeißen, die an einem bereits gesehenen Weg enden (die **filter**-Zeile). Auch sorgen wir mit **nubBy** dafür, dass nur Wege in der Liste bleiben, die in verschiedene Zustände enden. Diese Liste wird nun zusammen mit einer aktualisierten Liste passierter Zustände wiederrum an `seen'` gefüttert.

Das **reverse** sorgt dafür, dass der Startknoten am Anfang und das Ergebnis am Ende des Lösungsweges stehen.

```
74 graphSolve start next check = reverse $ solve' [[start]] []
75         where solve' l seen = case find (check.head) l of
76                   Just way → way
77                   Nothing  → solve' l' seen'
78                              where seen' = (map head l) ++ seen
79                                    l'    = nubBy sameHead $
80                                            filter (\w → (head w) `notElem` seen') $
81                                            concatMap next' l
```

```
82          next' (s:l) = map (:s:l) $ next s
83          sameHead l1 l2 = head l1 == head l2
```

Das wars auch schon, wir müssen nur noch `graphSolve` auf unsere Startkonfiguration los lassen und ihr dabei die passenden Parameter übergeben, und können die zurückgegebene Liste dann direkt ausgeben.

```
84 main = hPutStr stdout $ show' 1 $ graphSolve start moves solved
```

3 Finito

Natürlich bin ich über Fehlerberichte, Ergänzungen, Verbesserungen, Kommentare und Lob jederzeit empfangsbereit, am besten per e-Mail an `mail@joachim-breitner.de`.

Letze Änderung: May 14, 2006 19:10:46Z