Tutor: Martin Ring
Gruppe: 1
Tobias Brandt
Stefan Heitmann

Übungsblatt 1

Aufgabenlösung
Abgabe: 27.04.2017

1.1 *Labyrinthe erzeugen*

Cellular.hs

Ein zellulärer Automat ist bei uns ein *State Monad Transformer*, der ein Board hält. Ein Board ist bei uns einfach eine Map von Koordinaten auf den Status der jeweiligen Zelle.

```
9  data Status = Alive | Dead deriving (Eq, Show)
10 type Cell = (Int, Int)
11 type Board = M.Map Cell Status
12 type CA a = S.StateT Board IO a
```

Regeln werden bei uns als eine Funktion repräsentiert, die eine Zelle und ein Board entgegennimmt und einen entsprechenden neuen Status für diese Zelle liefert.

```
13 type Rule = (Cell → Board → Status)
```

Die Maze-Regel ist demnach ebenfalls eine Funktion.

```
15 nearCells :: Cell → [Cell]
16 nearCells (x,y) = [(x',y') | x' ← [x−1..x+1], y' ← [y−1..y+1]]
17
18 nearCellsStatus :: Cell → Board → [Status]
19 nearCellsStatus c m = catMaybes $ (`M.lookup` m) <$> nearCells c
20
21 countNearAliveCells :: Cell → Board → Int
22 countNearAliveCells c = length ∘ filter (==Alive) ∘ nearCellsStatus c
23
24 -- Assumption: c is a key within m
25 rule :: Rule
26 rule c m = case M.lookup c m of
27   Just Alive → if countNearAliveCells c m `elem` [1..5] then Alive else Dead
28   Just Dead  → if countNearAliveCells c m == 3 then Alive else Dead
29   Nothing    → error "Unexpected Error"
```

Die Funktion *initialState* erzeugt einen zufälligen Anfangsstatus des Automaten. Die *inner monad* des zellulären Automaten, ermöglicht es uns dabei die IO-Aktion *randomRIO* direkt zu benutzen.

```
44 initialState :: Int → CA ()
45 initialState s = do
46   n ← S.lift $ R.randomRIO (0,s*s)
47   [xs,ys] ← S.lift ∘ S.replicateM 2 ∘ S.replicateM n $ R.randomRIO (0,s)
48   let living = M.fromList $ zipWith (λx y → ((x,y),Alive)) xs ys
49   let rest = M.fromList [((x,y), Dead) | x ← [0..s], y ← [0..s]]
50   S.put $ M.union living rest
```

stepCellular wendet eine Regel auf den gesamten Automaten an.

```
31 stepCellular :: Rule → CA [(Cell, Status)]
32 stepCellular r = do
33   st ← S.get
34   let st' = [(c, r c st) | c ← M.keys st]
35   S.put $ M.fromList st'
36   return st'
```

Die Funktion *convergeCellular* wendet *stepCellular* solange an, bis der Automat konvergiert.

```
38 convergeCellular :: Rule → CA ()
39 convergeCellular r = do
40   st ← M.toList <$> S.get
41   st' ← stepCellular r
42   unless (st == st') $ convergeCellular r
```

Um später den Automat leichter von außen benutzen zu können, wurden die Hilfsfunktion *dead* und *runCA* implementiert.*dead* gibt Auskunft darüber, ob eine Zelle tot ist oder nicht. *runCA* startet den Automaten und gibt den zuletzt gültigen Board-Zustand zurück.

```
52 dead :: Cell → Board → Bool
53 dead c m = case M.lookup c m of
54   Just Dead → True
55   _         → False
56
57 runCA :: Int → IO Board
58 runCA s = S.execStateT (initialState s ≫ convergeCellular rule) undefined
```

Maze.hs

Um neue Labyrinthe zu erzeugen, wurde die Funktion *newMaze* so angepasst, dass sie alle Zellen, die nicht tot sind als *blocked* ansieht.

```
43 newMaze :: Int → IO Maze
44 newMaze s = do
45   board ← runCA s
46   return ∘ Maze s s $ λp → not $ dead (x p, y p) board
```

Um eine (einigermaßen) geeignete Startposition zu suchen, wurde die Funktion *freeCell* implementiert, die von einer gegeben Position aus, die am nähsten gelegende freie Zellenposition zurückgibt.

```
57 freeCell :: Int → Position → (Position → Bool) → Position
58 freeCell s pos p = case [Position x y | x ← [0..s], y ← [0..s], not ∘ p $ Position x y] of
59   []  → Position 0 0
60   xs → minimumBy (compare `on` dist pos) xs
61
62 dist :: Position → Position → Int
63 dist (Position x1 y1) (Position x2 y2) = round ∘ sqrt ∘ fromIntegral $ (x2–x1)^2 + (y2–y1)^2
```

Die Funktion *newGame* wurde so angepasst, dass sie die zuvor beschriebenen Funktionen aufruft und einen neuen GameState zurückgibt.

```
48 newGame :: Int → IO GameState
49 newGame s = do
50   mz ← newMaze s
51   return GameState {
52     maze = mz,
53     position = freeCell s (Position 0 0) $ blocked mz,
54     target = freeCell s (Position (s−1) (s−1)) $ blocked mz
55   }
```

Main.hs

Innerhalb von *startGame* wurde das *let-binding* in ein monadisches Binding geändert. Ebenso wurde der Zufallsgenerator entfernt.

```
18 startGame :: WS.Connection → IO ()
19 startGame conn = do
20   let send msg = WS.sendTextData conn (encode msg)
21   -- Initialisiere einen Zufallsgenerator
22   -- Erstelle ein Spiel mit einem 24x24 großen Labyrinth
23   game ← newGame 24
24   -- Schicke den initialen Spielzustand an den Client
25   send $ NewGame game
26   -- Warte auf Anweisungen vom Client
27   handleSocket conn game
```

1.2 *Labyrinthe lösen*

Main.scala

Das Lösen der Labyrinthe wurde mithilfe einer Breitensuche auf einer Warteschlange einer Richtungssequenz zum Lösen des Labyrinthes implementiert, welche terminiert, sobald sie den kürzesten Pfad findet. Diese wird innerhalb der Hilfsfunktionen *solveMaze*, welche wiederum *solve* enthält, aufgerufen.
solveMaze führt intern das Labyrinthlösen unter Berücksichtigung der Start- und Endposition sowie des Labyrinthes aus und verarbeitet dessen Lösung:

```
62     def solveMaze(maze: Maze, start: Position, target : Position) : Unit = {
63       solve(maze, start, target) match {
64         case None ⇒ window.alert("Maze not solveable")
65         case Some(path) ⇒ sendPath(path)
66       }
67     }
```

solve initialisiert die Warteschlange sowie eine für den Algorithmus notwendige Menge an besuchten Positionen:

```
70     def solve(maze : Maze, start : Position, target : Position): Option[Seq[Direction]] = {
71       val visited = Set[Position]()
72       val q = Queue[(Seq[Direction], Position)]((Seq(),start))
73       shortestPath(q, visited, target, maze)
74     }
```

Der Hauptalgorithmus der Berechnung des (kürzesten) Pfades *shortestPath* sieht folgendermaßen aus:

```
76      def shortestPath(q : Queue[(Seq[Direction], Position)], visited : Set[Position],
77       target : Position, maze : Maze) : Option[Seq[Direction]] = {
78        q match {
79        case q : Queue[(Seq[Direction], Position)] if q.isEmpty ⇒ None
80        case _ ⇒
81          val ((path, cur), qq) = q.dequeue
82          if (cur == target) {
83            Some(path)
84          } else if (visited contains cur) {
85            shortestPath(qq, visited, target, maze)
86          } else {
87            val newVisited = visited + cur
88            val qqq = qq ++ (for (
89              (dir, pos) ← neighbours(cur, maze)
90            ) yield {
91              (path :+ dir, pos)
92            })
93            shortestPath(qqq, newVisited, target, maze)
94          }
95        }
96      }
```

Wir prüfen, ob die Warteschlange leer ist. Ist diese leer, bedeutet es, dass es keinen Weg gibt und es wird zurückgegeben, dass es keine Lösung gibt (durch None).
Ist die Warteschlange nicht leer, so prüfen wir zuerst, ob die momentan betrachtete Position die Endposition ist. Ist sie die Endposition, so geben wir die Richtungssequenz zu dieser Position aus.
Wenn wir diese Position bereits besucht haben, so führen wir die Suche mit der nächsten Position fort.
Anderenfalls extrahieren wir den aktuell betrachteten Pfad (Richtungssequenz *newVisited*) der aktuell betrachteten Position, und fügen alle validen Nachbarpositionen mit dem jeweiligen Pfad von der aktuellen Position zur Warteschlange hinzu. Danach rufen wir *shortestPath* auf den aktualisierten Werten auf.

Interessant ist noch die Funktion zum Berechnen der Nachbarpositionen und dessen Richtung namens *neighbours*:

```
98      def neighbours(p : Position, maze : Maze) : Seq[(Direction, Position)] = {
99        for (i ← Seq.range(−1,2);
100            ii ← Seq.range(−1,2);
101            if (((i != 0 || ii != 0) && !(i != 0 && ii != 0)) &&
102              !(maze.blocked(Position(p.x+i, p.y+ii))) && p.x+i ≥ 0
103            && p.x+i < maze.width && p.y+ii ≥ 0 && p.y+ii < maze.height)
104        ) yield {
105            val pp = Position(p.x+i,p.y+ii)
106            (i, ii) match {
107              case (1,0) ⇒ (Direction.East, pp)
108              case (−1,0) ⇒ (Direction.West, pp)
109              case (0,−1) ⇒ (Direction.North, pp)
110              case (0,1) ⇒ (Direction.South, pp)
111              case _ ⇒ throw new Exception("should␣not␣happen")
112            }
113        }
114      }
115    }
```

Innerhalb des for-Teils wird lediglich eine simple XOR-Schaltung implementiert, welche genau die Richtungen von der aktuellen Positionen berechnet, die jeweils ein Feld zur Seite sind (nicht diagonal). Zusätzlich werden die Grenzen des Labyrinths beachtet und nur nicht blockierte Felder als Nachbarn anerkannt.
Für alle benachbarten Positionen wird danach die Richtungsbezeichnung von der aktuellen Position sowie

die Position selbst zurückgegeben.

Sollte ein Pfad gefunden worden sein, wird dieser dann mithilfe der *sendPath*-Funktion an den Server weitergeleitet.

```
52    def sendPath(path : Seq[Direction]) : Unit = {
53      path.headOption match {
54        case None ⇒ ()
55        case Some(d) ⇒ window.setTimeout(() ⇒ {
56          socket.send(encodeClientMessage(Client.Move(d)).noSpaces)
57          sendPath(path.tail)
58        },400)
59      }
60    }
```

1.3 *Tests*

Die erste Aufgabe wurde durch mehrmaliges Aktualisieren des Webbrowsers getestet. So haben wir feststellen können, dass beim Aktualisieren jedesmal ein neues Labyrinth generiert wird.

Da der Server durch die *newGame*-Funktion es ermöglicht, verschiedene Labyrinthgrößen zu erzeugen, haben wir während des Testens die Größe verkleinert, sodass bei zureichender Größe Labyrinthe erzeugt worden sind, bei denen es einen Pfad vom Start zum Ende gab. Der kürzeste Weg wurde durch unseren Pfadealgorithmus in allen bekannten Fällen gefunden.