Christoph Lüth
Marcus Ermler
Christian Maeder
Martin Ring
Diedrich Wolter

# 10. Übungsblatt

Ausgabe: 08.01.13
Abgabe: 18.01.13

Zu Beginn der Vorlesung *Praktische Informatik 3* haben wir als Beispiel einen Datentyp zur Repräsentation eines Labyrinths kennengelernt. In diesem Übungsblatt wollen wir ein Labyrinth zufällig erzeugen und erkunden.

## 10.1 *In der Zelle* — *7 Punkte*

Für die Generierung des zufälligen Labyrinths nutzen wir einen begrenzten quadratischen *zellulären Automaten*, dessen Zustand aus einem zweidimensionalem Raster von Zellen besteht, deren Zustand entweder 'lebendig' oder 'tot' ist.

Für den Zustandsübergang des Automaten wird für jede Zelle und ihre acht direkten und diagonalen Nachbarn die Summe der lebenden Zellen ermittelt. Eine Übergangsregel spezifiziert, bei wievielen Nachbarn eine Zelle überlebt bzw. geboren wird, und wann eine Zelle tot ist bzw. stirbt.

Ein bekannter zellulärer Automat ist *Conway's Game Of Life*. Zur Erzeugung von Labyrinthen verwenden wir ähnliche Automaten mit der sogenannten *Maze*-Regel:

- Eine Zelle überlebt genau dann, wenn sie lebendig ist und zwischen 1 und 5 lebendige Nachbarn hat.
- Eine Zelle wird geboren (der Zustand ändert sich zu lebendig), wenn sie genau 3 lebendige Nachbarn hat.
- Ansonsten bleibt die Zelle tot, oder stirbt.

Implementieren sie ein Modul `Cellular` mit folgenden Funktionen:

- Entwerfen Sie eine geeignete Repräsentation für zelluläre Automaten.
- Entwerfen Sie eine allgemeine Repräsentation von Zustandsübergangsregeln, und geben Sie eine Repräsentation der *Maze*-Regel an.
- Schreiben sie eine Funktion `initialState size` welche als initialen Zustand ein zufällig gefülltes quadratisches Raster mit Kantenlänge `size` erzeugt. Welche Signatur muss `initialState` haben? Warum?
- Implementieren Sie eine Funktion `stepCellular rule`, die einen Zustandsübergang des zellulären Automaten mit der Regel `rule` ausführt.
- Zu guter Letzt implementieren sie eine Funktion `convergeCellular rule`, welche so lange einen Zustandsübergang des zellulären Automaten mit der angegeben Regel durchführt bis ein stabiler Zustand erreicht ist (d.h. keine Veränderung mehr stattfindet). Sie können davon ausgehen, dass die *Maze*-Regel immer konvergiert (auch wenn das in der Praxis nicht immer zutrifft).

## 10.2 *Labyrinth* — *7 Punkte*

Das eigentliche Labyrinth und Positionen darin werden durch einen abstrakten Datentyp in einem Modul `Labyrinth` repräsentiert, dessen Schnittstelle wie folgt ist:

- Der Datentyp `Direction` repräsentiert Richtungen:

  ```
  data Direction = North | West | South | East
  ```

- Der abstrakte Typ Labyrinth repräsentiert ein Labyrinth. Er soll Instanz von Read und Show sein, aber die textuelle Repräsentation muss nicht verständlich sein (dazu siehe display unten).

  ```
  data Labyrinth

  instance Read Labyrinth where...
  instance Show Labyrinth where...
  ```

- Der abstrakte Typ Position repräsentiert *Positionen* im Labyrinth. Er soll Instanz von Ord sein. start und exit geben Start- und Endposition im Labyrinth zurück, und possibilities die von einer Position aus erreichbaren Nachbarpositionen.

  ```
  data Position
  instance Ord Position where...

  start :: Labyrinth → Position
  exit  :: Labyrinth → Position

  possibilities :: Labyrinth → Position → [(Direction,Position)]
  ```

- display gibt eine verständliche Repräsentation des Labyrinths aus, wobei die übergebene Position gekennzeichnet werden soll:

  ```
  display :: Labyrinth→ Position→ String
  ```

- generate erzeugt mit Hilfe eines zellulären Automaten aus der vorherigen Aufgabe ein neues Labyrinth der vorgegebenen Größe:

  ```
  generate :: Int → IO Labyrinth
  ```

Zum Testen Ihres Labyrinths können Sie das vorgegebene Solve-Modul verwenden, das eine Funktion shortestPath bietet, welche den kürzesten Pfad von Start- zu Endposition sucht. Mit instructions kann das Ergebnis lesbar angezeigt werden. Beachten Sie, dass die mit der *Maze*-Regel erzeugten Labyrinthe nicht immer lösbar sein müssen (das brauchen Sie nicht zu beheben).

## 10.3 *Interaktion* — *6 Punkte*

Jetzt soll der Benutzer das Labyrinth interaktiv erkunden können! Die Erkundung beginnt an der Startposition. Es werden dem Nutzer die möglichen Richtungen anzeigt, und dann eine Eingabe abgewartet, welche die nächste Position bestimmt. Die Erkundung endet, wenn der Benutzer das Ziel erreicht (welches mit einer geeignet freudigen Botschaft signalisert werden soll). Die Eingabe soll außer den möglichen Richtungen auch die Möglichkeit bieten, den momentanen Spielzustand abzuspeichern, und das Labyrinth mit der momentanen Position darin anzuzeigen. Beachten Sie hierbei, dass nicht jeder Nutzer immer fehlerfreie Eingaben liefert, das soll nicht zu einem Programmabbruch führen.

Um den Spielzustand — bestehend aus dem Labyrinth und der momentanen Position — persistent speichern zu können, implementieren Sie folgende zwei Funktionen, welche diese aus einer Datei lesen bzw. in eine schreiben

```
load :: FilePath → IO (Labyrinth, Position)
save :: FilePath → IO ()
```

Der Labyrinthexplorer soll von der Kommandozeile aus aufgerufen werden können; ein Kommandozeilenargument gibt die Größe des neu zu generierenden Labyrinths an, oder den Namen einer Datei, aus dem der Spielzustand geladen werden soll.

## ? *Verständnisfragen*

1. Warum ist die Erzeugung von Zufallszahlen eine Aktion?
2. Warum sind Aktionen nicht explizit als Zustandsübergang modelliert, sonder als abstrakter Datentyp IO?
3. Außer dem Mangel an referentieller Transparenz, was ist die entscheidende Eigenschaft, die Aktionen von reinen Funktionen unterscheidet?