Reaktive Programmierung SoSe 19

Christoph Lüth
Martin Ring

1. Übungsblatt

Ausgabe: 11.04.19
Abgabe: 02.05.19

In diesem Übungsblatt geht es um Labyrinthe — „a maze of twisty little passages, all alike“. Dabei wollen wir zum einen Labyrinthe zufällig erzeugen, und zum anderen Wege durch (solche zufällig erzeugten) Labyrinthe suchen. Das Gesamtpaket ist dann unser *electronic labyrinth simulator and explorer* (ELSE).

Um die beiden in dieser Veranstaltung verwendeten Sprachen im direkten Vergleich zu sehen, implementieren wir die Generierung in Haskell, und die Wegsuche in Scala. Wir haben für ELSE eine Rahmenwerk vorgegeben, welches die interaktive Exploration von erzeugten Labyrinthen erlaubt. Das im cutting edge der Webtechnologien implementierte Rahmenwerk besteht aus zwei Teilen (Abb. 1). Der Server ist in Haskell implementiert, und verwaltet das Labyrinth. Mit der *warp* Server-Bücherei ermöglicht er dem Client Zugriff auf das Labyrinth. Der Client ist in Scala implementiert, wird mit Scala.js zu JavaScript übersetzt und läuft dann in einem Webbrowser. Scala und Haskell kommunizieren, indem sie Daten — insbesondere eine Repräsentation des Labyrinths — per JSON über einen Websocket austauschen.

Sie brauchen sich um den technischen setup in diesem Übungsblatt keine Gedanken zu machen, sondern können sich auf die folgenden zwei Aufgaben konzentrieren, indem sie die auf der Webseite verfügbare Vorgabe von ELSE an zwei strategischen Stellen erweitern.

1.1 *Labyrinthe erzeugen* — *10 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.

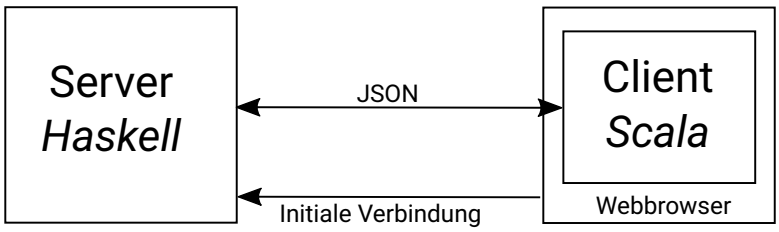

Abbildung 1: Systemarchitektur von ELSE.

- 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).

Das eigentliche Labyrinth und Positionen darin werden durch einen abstrakten Datentyp in einem Modul Maze durch den gleichnamigen Datentyp repräsentiert:

```
data Maze = Maze { width :: Int
                 , height :: Int
                 , blocked :: Position → Bool
                 }
```

Die Funktion newMaze :: RandomGen g⇒ g→ Int→ Int→ Maze erzeugt ein neues Labyrinth; erweitern Sie newMaze so, dass es zelluläre Automaten benutzt, um zufällig ein Labyrinth zu erzeugen.

Erzeugen sie außerdem in der Funtion newGame eine geeignete freie Start- und Zielposition (möglichst weit links/oben bzw. rechts/unten).

1.2 *Labyrinthe lösen* *10 Punkte*

Implementieren Sie Clientseitig in der Klasse Maze, welche Labyrinthe repräsentiert, eine Methode

```
def solve() :: Option[Seq[Direction]]
```

welche den *kürzesten* Weg durch das Labyrinth sucht, wenn es einen gibt.

Integrieren Sie ihre Lösung in den vorgegebenen Client, indem der Client aus dem vom Server übertragenen Labyrinth den kürzesten Weg sucht, und diesen der Reihe nach abläuft, indem es entsprechende Kommandos an den Server schickt. (Der Benutzer braucht in diesem Fall gar nichts mehr zu tun.)

Um das Spiel besser genießen zu können, können sie die JavaScript Funktion window.setTimeout(f,n) verwenden um eine Funktion f nach n millisekunden auszuführen und damit eine Verzögerung zwischen den Kommandos erzeugen.

Hinweis: Sie können die Lösung von Aufgabe 1.1 unabhängig von Aufgabe 1.2 testen, indem Sie einfach versuchen, das Labyrinth interaktiv zu testen. Auf der anderen Seite können Sie auch die Aufgabe 1.2 testen, indem Sie den Server einige fest vorgebenen Labyrinthe servieren lassen.