Universität Bremen

Reaktive Programmierung SoSe 15

Christoph Lüth
Martin Ring

1. Übungsblatt

Ausgabe: 23.04.15
Abgabe: 11.05.15

Vorwort

Wir schreiben das Jahr 2137. Raumfahrt ist so alltäglich wie Bahnfahren heute (die Gewerkschaft der Raketenpiloten hat gerade einen neuen Streik angekündigt), und da die Ressourcen auf der Erde weitgehend erschöpft sind, wird auf Asteroiden und den anderen Planeten nach Mineralien und Erzen geschürft.

Dieses und die folgenden Übungsblätter führen uns die Welt der Roboter-Minen auf dem Asteroiden XZ340. Hier bauen fern der Erde Roboter in einer lebensfeindlichen Umgebung Gold ab. In der Mine finden sich ferner Dilithium-Kristalle, mit welchen die Roboter ihre Akkus aufladen können. Außerdem enthält die Mine natürlich alle Arten von Steinen, sonstigen Hindernissen und weiteren Überraschungen. Wir wollen am Anfang einen (später mehrere) Roboter in diesen Minen interaktiv oder autonom steuern.

1.1 *Es gibt Punkte!* — *5 Punkte*

Die Mine wird durch ein Gitter von Objekten beschrieben. Das Gitter wird durch Punkte der Form (x, y) indiziert, daher fangen wir mit der Modellierung von Punkten an. Diese werden als algebraischer Datentyp

```
case class Point(x: Int, y: Int)
```

modelliert. Wir benötigen für diesen Methoden zur Addition, Subtraktion, Konversion in Polarkoordinaten (durch zwei Methoden, die zum einen die Länge und zum anderen den Winkel des Punktvektors berechnen).

Implementieren Sie ferner folgende zwei Methoden:

- `env(r: Int): Set[Point]` gibt die Menge aller Punkte in einem Rechteck der Kantenlänge $2r + 1$ um den Punkt herum;
- `shadowed(q: Point)(r: Point): Boolean` gibt an, ob der Punkt R bezüglich des Punktes Q *verdeckt* wird. Ein Hindernis in Q verdeckt dabei einen anderen Punkt R bezüglich eines Punktes P (i.e. `this`), wenn die direkte Linie vom Mittelpunkt von P zum Mittelpunkt von R durch das Hindernis in Q führt; wir nehmen an, dass das Hindernis in Q das gesamte Gitterfeld ausfüllt. Dabei soll Q sich nicht selbst verdecken.

1.2 *Ab in die Mine* — *5 Punkte*

Die Objekte der Mine sind ein algebraischer Datentyp der Form

```
trait Obj
case class Gold(value: Int) extends Obj
case class Dilithium(nrg: Int) extends Obj
case object Base extends Obj
case object Empty extends Obj
case object Obst extends Obj
```

Wir abstrahieren die Mine als ein Gitter, welches solche Objekte enthält:

```
class Mine(width: Int, height: Int, world: Map[Point, Obj])
```

Implementieren Sie für diesen Datentyp folgende Methoden:

- `get(p: Point): Obj` und `update(p: Point, o: Obj)`, um das Objekt an einer Position abzufragen oder zu ändern;

- Ein Konstruktor `this(w: Int, h: Int)`, welcher eine Mine der angegebenen Größe erstellt, und diese zufällig mit einer Anzahl Hindernissen, Gold und Dilithiumkristallen füllt. (Deren Anzahl kann als fester Prozensatz der Größe der Mine angegeben werden.)
- Eine Methode `radar(p: Point, r: Int): Set[Point]`, welche die von dem angegeben Punkt `p` in der Entfernung `r` *sichtbaren* (d.h. nicht durch Hindernisse verdeckten) Punkte angibt.

1.3 *Das große Ganze* — *10 Punkte*

Jetzt modellieren wir noch den Roboter und den gesamten Systemzustand. Der Roboter hat als Zustand eine Position, Batterieladezustand und gesammeltes Gold

```
case class Robot(pos: Point, energy : Int, gold : Int)
```

sowie Methoden, die den Roboter bewegen, ihn Gold und Dilithiumkristalle aufnehmen lassen.

Der gesamte Systemzustand besteht aus der Mine und dem Roboter, und implementiert zwei wichtigen Methoden:

```
class State(w: Mine, r: Robot)
```

- `render(): String` erzeugt eine textuelle Repräsentation des Zustandes (s.u.), und
- `moveRobot(c: Control): Option[(State, String)]` führt das Kommand `c` aus, und gibt einen Folgezustand zurück (oder `Nothing`, falls der Benutzer die Simulation abbricht).

Ein Kommando, repräsentiert durch den algebraischen Datentyp `Control`, ist dabei entweder die Bewegung in eine gegebene Richtung (N, NE, E, SE, S, ...), das Aufsammeln von Gold oder Dilithium, oder das Spielende.

Die Hauptfunktion der Robotersimulation soll eine konsolenbasierte interaktive Simulation ermöglichen (im Stil von Nethack[1]). Hierbei werden die im Radar (`radar`, siehe oben) sichtbaren Felder sowie der Zustand des Roboters angezeigt, und der Benutzer zu einer Eingabe eines Kommandos (bspw. `"n"` für „Gehe nach Norden“) aufgefordert. Nach der Eingabe eines gültigen Kommandos wird dieses ausgeführt, der neue Zustand visualisert etc. Abb. 1 zeigt eine mögliche textuelle Darstellung.

Die Simulation endet

- *erfolglos*, wenn der Roboter keine Energie mehr hat,
- oder *erfolgreich*, wenn der Roboter wieder zurück in die Heimatbasis (das Feld, von dem aus der startet) kommt.

```
..  ..*     ..
 ..  ... .  .
  .!*... .*..
   ..... .!.
    .!$.*..$*
     ..B.!...
   * *......*   .
.......$.....$..!
.......$X....!..*
.$....$$*!..*
 *..!!   .$.
  .$..     ....$
... *       * $..
..              .$
.                .
[Pos: Point(9,11)]---[$$$: 10]---[NRG: 15]
Picked up 1 gold.
COMMAND>
```

Abbildung 1: Textuelle Visualisierung des Simulationszustandes. `$` ist Gold, `!` Dilithium, `X` die Position des Roboters, `*` Hindernisse und `B` die Heimatbasis.

Hinweise:

- Benutzen Sie, wenn möglich, den funktionalen Subset von Scala. Jede Verwendung von veränderlichen Variablen oder Feldern (`var`) oder `while`-Schleifen führt leider zu einem Abzug von einem halben Punkt.
- Auf der Webseite wird eine Datei `build.sbt` zur Verfügung gestellt, welche die Verwendung von `sbt`[2] für diese Aufgabe ermöglicht.

[1] `http://www.nethack.org/`
[2] `http://www.scala-sbt.org/`