Reaktive Programmierung SoSe 19

Christoph Lüth
Martin Ring

2. Übungsblatt

Ausgabe: 02.05.19
Abgabe: 16.05.19

2.1 *Promise: My Future in Haskell* — *3 Punkte*

In dieser Aufgabe wollen wir Futures und Promises wie wir sie in Scala kennengelernt haben in Haskell implementieren. Dazu implementieren wir ein Modul Future mit den folgenden Funktionen:

```
data Promise α
data Future α

promise   :: IO (Promise α)                  -- Erzeugt ein Promise
complete  :: Promise α → α → IO ()           -- Erfüllt ein Promise

future    :: Promise α → Future α            -- Die Future zu einem Promise
onComplete :: Future α → (α → IO ()) → Future α
wait      :: Future α → IO α                 -- Wartet, dass die Future erfüllt wird
```

Dabei sind Future und Promise komplett asynchron; es wird nirgendwo ein neuer Thread erzeugt.

Das Beispiel Robots.hs (in der Vorlage enthalten) zeigt die Verwendung von Future und Promise mit der oben angegebenen Schnittstelle; es sollte sich bei korrekter Implementierung wie das Scala-Beispiel aus der Vorlesung verhalten.

2.2 *MVar in Scala?!* — *2 Punkte*

Nachdem wir Future in Haskell implementiert haben, wollen wir den anderen Weg gehen und MVar in Scala implementieren. Oder auch nicht — was ist das große Problem bei einer Implementation von MVar in Scala?

Skizzieren Sie eine Lösung (welche Schnittstelle hätte eine Scala-Implementierung), und schildern Sie die auftretenden Probleme.

2.3 *Liften lassen? Nicht in diesem Aufzug!* — *15 Punkte*

Aufzugssteuerung ist tatsächlich eine hochgradig nicht-triviale Angelegenheit, deren Komplexität man auf den ersten (und auch zweiten Blick) nicht so recht erfasst. Hier werden wir eine responsive Lösung für dieses Problem entwickeln.

In dem Repository `https://github.com/martinring/elevator` finden sie eine Aufzugsimulation, welche innerhalb einer SVG Datei implementiert ist. Außerdem gibt es je eine Vorlage in Scala und Haskell für die Implementierung einer Aufzugsteuerung, von denen Sie eine auswählen können. Die SVG-Datei und ihre Implementierung kommunizieren über JSON Nachrichten. Die (De-)Serialisierung der Nachrichten ist in der Vorlage bereits implementiert. Wie sie die Simulation mit der Steuerung verbinden können entnehmen Sie der entprechenden `README` Datei.

Folgene Nachrichten muss ihre Implementierung verarbeiten können:

- `Initialize`: Diese Nachricht wird als erstes von der Simulation gesendet und beinhaltet Informationen über die Anzahl der Stockwerke sowie die Anzahl der Aufzugschächte und die Kapazitäten der Aufzüge.
- `ElevatorBroken`: Diese Nachricht wird gesendet, wenn ein Aufzug durch falsche Steuersignale zerstört wurde. Ihre Implementierung sollte vermeiden, dass dieser Fall eintritt.

- `ElevatorDoorBroken`: Die Tür eines Aufzugs wurde durch falsche Steuersignale zerstört.
- `FloorDoorBroken`: Die Tür auf einer Etage wurde durch falsche Steuersignale zerstört.
- `FloorPassed`: Ein Aufzug ist an einem Kontrollpunkt genau zwischen zwei Etagen vorbei gefahren.
- `ElevatorStopped`: Ein Aufzug hatt an einer Etage angehalten.
- `ElevatorDoorOpened`: Eine Aufzugtür wurde vollständig geöffnet.
- `ElevatorDoorClosed`: Eine Aufzugtür wurde vollständig geschlossen.
- `FloorDoorOpened`: Eine Etagentür wurde vollständig geöffnet.
- `FloorDoorClosed`: Eine Etagentür wurde vollständig geschlossen.
- `ElevatorButtonPressed`: Ein Auswahlknopf für eine Etage in einem Aufzug wurde betätigt.
- `FloorButtonPressed`: Ein Auswahlknopf für eine Fahrtrichtung wurde auf einer Etage betätigt.

Folgende Steuersignale können sie an die Simulation senden. Die Bedingungen müssen dabei jeweils erfüllt sein.

- `MoveElevator`: Bewegt den Aufzug in eine Richtung.
  Bedingungen: Sowohl die Aufzugtür als auch die Etagentür müssen geschlossen sein; der Aufzug darf nicht in Bewegung sein.
- `StopElevator`: Lässt den Aufzug an der nächsten Etage anhalten.
  Bedingung: Der Aufzug muss in Bewegung sein.
- `OpenElevatorDoor`: Öffnet eine Aufzugtür.
  Bedingungen: Die Aufzugtür muss vollständig geschlossen sein; der Aufzug darf nicht in Bewegung sein.
- `OpenFloorDoor`: Öffnet eine Etagentür.
  Bedingungen: Die Etagentür muss vollständig geschlossen sein; der Aufzug muss stehen und sich hinter der Etagentür befinden.
- `CloseElevatorDoor`: Schließt eine Aufzugtür.
  Bedingung: Die Aufzugtür muss vollständig geöffnet sein.
- `CloseFloorDoor`: Schließt eine Etagentür.
  Bedingung: Die Etagentür muss vollständig geöffnet sein.
- `SetElevatorButtonLight`: Schaltet die Beleuchtung eines Knopfes im Aufzug an oder aus.
  Bedingungen: Der Knopf wurde gedrückt, bzw. die Anfrage wurde bedient. Wenn der Knopf beleuchtet wird, darf sich der Aufzug nicht auf der entsprechenden Etage befinden.
- `SetFloorButtonLight`: Schaltet die Beleuchtung eines Knopfes auf der Etage an oder aus.
  Bedingungen: Der Knopf wurde gedrückt, bzw. die Anfrage wurde bedient. Wenn der Knopf beleuchtet wird, darf sich kein vollständig geöffneter Aufzug mit der entsprechenden Fahrtrichtung auf der Etage befinden.

In der Simulation bewegen sich Personen. Die Personen wollen sich möglichst schnell zufällig durch das Gebäude bewegen. Je größer die Wartezeiten desto unzufriedener (roter) werden die Personen. Ihre Implementierung soll die Personen möglichst zufrieden (grün) halten.

Die Zustände der Aufzüge können Sie nicht explizit erfragen. Stattdessen müssen Sie diese aus den Nachrichten selbst herleiten. Am Anfang befinden sich jedoch alle Aufzüge in der untersten Etage (0), alle Türen sind geschlossen und alle Knöpfe sind unbeleuchtet.

Ihre Implementierung darf nicht blockieren, da die Responsivität der Steuerung sehr wichtig ist: Wenn Sie einen Aufzug nicht rechtzeitig stoppen kann er zerstört werden, wenn er sich über das oberste oder das unterste Stockwerk hinaus bewegt (was Sie natürlich vermeiden sollten).

Steigern Sie die Größe und Geschwindigkeit der Simulation und beobachten Sie, ob Ihre Implementierung sich weiter korrekt verhält. Wann tut sie es nicht mehr und warum?

Happy Lifting!