3. Übungsblatt

Ausgabe: 02.12.09
Abgabe: 16.12.09

In diesem Übungsblatt werden wir die Nebenläufigkeitsabstraktion der hybriden Programmiersprache Scala (bekannt aus Presse, Funk und Fernsehen) in Haskell nachbilden. Es handelt sich dabei um sogenannte *Aktoren* [1].

Die Scala-Dokumentation findet sich hier: `http://www.scala-lang.org/api`.

5 *Nachrichtenschlangen* *7 Punkte*

Die Basiskommunikation zwischen Aktoren wird durch *message queues* (Nachrichtenschlangen) implementiert. Eine Nachrichtenschlange ist ähnlich den bereits bekannten Kanälen (`Control.Concurrent.Chan`), erlaubt aber zusätzlich noch den *synchronen* Versand von Nachrichten: synchron heißt hier, das der sendende Thread so lange wartet, bis die gesendete Nachricht gelesen worden ist.

Die Signatur ist damit wie folgt:

```
data MQ a

newMQ      :: MQ a  -- erzeugt leer Schlange
sendSync   :: MQ a → a → IO ()  -- synchrones Senden
sendAsync  :: MQ a → a → IO ()  -- asynchrones Senden
recv       :: MQ a → IO a       -- Lesen
```

6 *Aktoren* *13 Punkte*

Ein Aktor ist sozusagen ein Zustandsthread mit einer ihm zugeordneten Nachrichtenschlange, in Haskell natürlich als Monade definiert. Diese Monade hat folgende Operationen:

- an diesen Aktor gesendete Nachrichten lesen;
- an einen anderen Aktor Nachrichten senden;
- IO-Aktionen ausführen.

Aktoren sind parametrisiert über einen Typen a, welcher den Typen der Nachrichten repräsentiert, die dieser Actor empfangen kann.

Technisch unterscheiden wir zwischen dem Monaden Acting, der das Verhalten des Aktors definiert (sozusagen die "Rolle"), und dem abstrakten Datentypen Actor, der den Aktor repräsentiert.

[1] Wir verzichten auf die im Deutschen in diesem Kontex nicht gebräuchliche Übersetzung "Schauspieler".

```
data Acting a b

instance Monad (Acting a)
instance MonadIO (Acting a)         -- gibt Funktion liftIO

(!)    :: Actor a→ a→ Acting a () -- send (sync.)
(!?)   :: Actor a→ a→ Acting a () -- send (async.)
recv   :: Acting a a               -- receive
```

Erst wenn wir den Aktor starten, erhalten wir den eigentlichen Aktor:

```
act      :: Acting a ()→ IO (Actor a)
stop     :: Actor a→ IO ()
```