**Tutor**: Martin Ring
**Gruppe**: 1
Tobias Brandt
Stefan Heitmann

# Übungsblatt 2

Aufgabenlösung
Abgabe: 11.05.2017

---

## 2.1 *Promise: My Future in Haskell*

Ein Promise ist in Haskell nur ein Halter für ein Future. Ein Future hält eine MVar und eine mögliche Aktion.

```
data Promise a = Promise (Future a)
data Future a = Future { var :: MVar a, handle :: Maybe (a → IO ()) }
```

Der Rest der Aufgabe wurde nach Spezifikation implementiert.

```
promise :: IO (Promise a)
promise = do
  var ← newEmptyMVar
  return ∘ Promise $ Future var Nothing

complete :: Promise a → a → IO ()
complete (Promise f) = putMVar (var f)

future :: Promise a → Future a
future (Promise f) = f

onComplete :: Future a → (a → IO ()) → Future a
onComplete f action = f {handle = Just action}

wait :: Future a → IO a
wait f = do
  a ← takeMVar $ var f
  case handle f of
    Just action → action a
    _           → return ()
  return a
```

Zum Testen wurde das mitgelieferte Roboter-Programm ausgeführt. Es verhielt sich ähnlich zu dem in der Vorlesung gezeigten Scala-Implementierung.

## 2.2 *MVar in Scala?!*

```
class MVar[A] {
    var value = null
    def takeMVar : A = { ... }
    def putMVar(a : A) : Unit = {...}
}
```

## 2.3 *6 nimmt!*

Wir habens uns dazu entschlossen, die gesamte Funktionalität auf dem Server zu realisieren. Sodass Programme wie *Telnet*als Client verwendet werden können.

In der Funktion `startServer` wird ein Socket geöffnet. Anschließend wird bereits ein Thread für die Handhabung des Spielens gestartet (`newGame`). Danach wird in diesem Thread nur noch auf ankommende Verbindungen gewartet.

```
29 startServer :: IO ()
30 startServer = withSocketsDo $ do
31   putStrLn "Starting␣server."
32   sock ← listenOn $ PortNumber 8000
33   inChan ← newChan
34   outChan ← newChan
35   players ← newMVar []
36   forkIO $ newGame inChan outChan players
37   handleSocket sock inChan outChan players
38
39 handleSocket :: Socket → Chan Msg → Chan Msg → MVar [(Id, Bool)] → IO ()
40 handleSocket sock inChan outChan ps = forever $ do
41   (handle, _, _) ← accept sock
42   inChan' ← dupChan inChan
43   forkIO $ identify handle inChan' outChan ps
```

Sollte eine neue Verbindung eingehen, wird ein neuer Thread gestartet, der für die Identifizierung einer Person zuständig ist. Dieser erfragt einen Namen von der eigehenden Verbindung. Sollte der eingegangene Name valide sein, wird diese Verbindung als neuer Spieler in die Spielerlisten-MVar eingetragen. Zusätzlich wird der Spieler als "noch nicht bereit" markiert.

```
45 identify :: Handle → Chan Msg → Chan Msg → MVar [(Id, Bool)] → IO ()
46 identify handle inChan outChan ps = fix $ λloop → do
47   hPutStrLn handle "Please␣enter␣your␣name"
48   msg ← readHandle handle
49   ps' ← takeMVar ps
50   if null msg && isJust (lookup msg ps') then do
51     putMVar ps ps'
52     loop
53   else if length ps' > 9 || (all snd ps' && length ps' > 1) then do
54     putMVar ps ps'
55     hPutStrLn handle $ "Sorry,␣" ++ msg ++ ",␣but␣there␣is␣currently␣a␣game␣in␣progress.␣Try␣again␣
56   else let id = msg in do
57     hPutStrLn handle $ "Welcome␣" ++ id ++ "!"
58     forkIO $ readClient handle inChan id
59     outChan' ← dupChan outChan
60     forkIO $ writeClient handle outChan' id
61     writeChan outChan' ("", id ++ "␣joined␣the␣session.")
62     putMVar ps $ (id,False):ps'
63     return ()
```

Zusätzlich wird für jeden akzeptierten Client jeweils ein Thread zum Einlesen von Eingaben und zum Ausgeben vom Nachrichten vom Server gestartet. Die Kommunikation wird dabei über Channels realisiert. Damit benutzen wir MVars indirekt, da Channels *"lediglich"* einen Stream von MVars darstellen. Die Channels werden dabei immer von dem Wurzel-Eingabe- bzw. Wurzel-Ausgabe-Channel dupliziert, welche ebenfalls in dem Spiele-Thread bekannt sind.

```
65 readClient :: Handle → Chan Msg → Id → IO ()
66 readClient handle ch id = forever $ do
67   msg ← readHandle handle
68   writeChan ch (id, msg)
69
70 readHandle :: Handle → IO String
71 readHandle handle = filterMsg <$> hGetLine handle where
```

```
72   filterMsg = filter (not ∘ flip elem ['λr','\n'])
73
74 writeClient :: Handle → Chan Msg → Id → IO ()
75 writeClient handle ch id = forever $ do
76   (id', msg) ← readChan ch
77   when (null id' || id==id') $ hPutStrLn handle msg
```

Der Spiele-Thread besteht aus zwei Phasen.

```
79 newGame :: Chan Msg → Chan Msg → MVar [(Id, Bool)] → IO ()
80 newGame inChan outChan ps = do
81   preGame inChan outChan ps
82   game inChan outChan ps
```

Die erste Phase überprüft, ob alle Spieler in der Spielerlisten-MVar als "bereit" markiert sind. Sollte dies der Fall sein, endet die erste Phase und die zweite Phase beginnt. Zusätzlich stellt die erste Phase Kommandos für den Client bereit, um Informationen über die anderen Spieler zu ergattern.

```
84 preGame :: Chan Msg → Chan Msg → MVar [(Id, Bool)] → IO ()
85 preGame inChan outChan ps = fix $ λloop → do
86   ps' ← readMVar ps
87   unless (all snd ps' && length ps' > 1) $ do
88     (id', msg) ← readChan inChan
89     let msg' = toLower <$> msg
90     when (msg' == "ready") $ do
91       ps' ← takeMVar ps
92       let ps'' = map (λe@(n,b) → if n == id' then (n, True) else e) ps'
93       putMVar ps ps''
94       unless (ps' == ps'') $ writeChan outChan ("",id' ++ " is ready to play.")
95     when (msg' == "help") $
96       writeChan outChan (id', "Available commands: 'ready', 'help', 'whoisready', 'whoisconnected'")
97     when (msg' == "whoisready") $ do
98       ps' ← readMVar ps
99       forM_ (map fst $ filter snd ps') $ writeChan outChan ∘ (id', ) ∘ (++ " is ready.")
100    when (msg' == "whoisonline") $ do
101      ps' ← readMVar ps
102      forM_ (map fst ps') $ writeChan outChan ∘ (id', ) ∘ (++ " is connected.")
103    loop
```

In der zweiten Phase findet das eigentliche Spiel statt. Dazu haben wir zunächst das Spielfeld und die Spieler als algebraische Datentypen dargestellt.

```
23 data Player = Player { name :: String, score :: Int, hand :: [Int] } deriving Show
24 data Table = Table { rows :: [[Int]] } deriving Show
25 data ST = ST { players :: [Player], table :: Table } deriving Show
26
27 instance Eq Player where (Player n _ _) == (Player m _ _) = m == n
```

Außerdem haben wir eine Reihe an Funktionen geschrieben, die den entsprechenden Spielzustand an die Clients ausgibt.

```
147 printScores :: Chan Msg → ST → IO ()
148 printScores out st = forM_ (players st) $ printScore out
149
150 printScore :: Chan Msg → Player → IO ()
151 printScore out p = writeChan out (name p, "Your current score is " ++ (show ∘ score) p ++ ".")
152
153 printHands :: Chan Msg → ST → IO ()
154 printHands out st = forM_ (players st) $ printHand out
```

```
155
156 printHand :: Chan Msg → Player → IO ()
157 printHand out p = do
158   writeChan out (name p, "Your␣hand:␣")
159   writeChan out (name p, unwords ∘ map show $ hand p)
160
161 printTable :: Chan Msg → ST → IO ()
162 printTable out st = forM_ (players st) $ printTableForPlayer out (table st)
163
164 printTableForPlayer :: Chan Msg → Table → Player → IO ()
165 printTableForPlayer out table p = do
166   writeChan out (name p, "The␣current␣Table:␣")
167   forM_ (rows table) $ printRow out p
168
169 printRow :: Chan Msg → Player → [Int] → IO ()
170 printRow out p row = writeChan out (name p, unwords ∘ map show $ row)
```

Die game-Funktion initialisiert das Spiel, startet es und spielt solange Spielrunden bis ein Spieler mindestens 66 Minuspunkte gesammelt hat.

```
105 game :: Chan Msg → Chan Msg → MVar [(Id, Bool)] → IO ()
106 game inChan outChan var = do
107   writeChan outChan ("", "Starting␣Game␣Round.")
108   initPlayers ← initialPlayers var
109   st ← dealOut initPlayers
110   varST ← newIORef st
111   fix $ λloop → do
112     st' ← readIORef varST
113     shuffled ← dealOut (players st')
114     writeIORef varST shuffled
115     st' ← readIORef varST
116     unless (any (≥66) $ (map score ∘ players) st') $ do
117       writeChan outChan ("", "═══════════════════════════════════════")
118       fix $ λloop' → do
119         st' ← readIORef varST
120         unless (all null $ (map hand ∘ players) st') $ do
121           printHands outChan st'
122           newState ← selectCards st' inChan outChan
123           writeIORef varST newState
124           loop'
125       roundState ← readIORef varST
126       printScores outChan roundState
127       loop
128   endState ← readIORef varST
129   let winner = minimumBy (compare `on` score) $ players endState
130   writeChan outChan ("", "The␣winner␣is␣" ++ name winner ++ "!␣Congratulations!")
131
132 initialPlayers :: MVar [(Id, Bool)] → IO [Player]
133 initialPlayers var = do
134   ps ← (map fst) <$> readMVar var
135   return $ map (\n → Player n 0 []) ps
136
137 dealOut :: [Player] → IO ST
138 dealOut ps = do
139   cards ← shuffleCards
140   let ps' = map (λi → (ps !! i) {hand = cards !! i}) [0..length ps−1]
141   let table = Table ∘ map (:[]) $ last cards
```

```
142   return $ ST ps' table
143
144 shuffleCards :: IO [[Int]]
145 shuffleCards = chunksOf 10 <$> shuffleM [1..104]
```

Eine einzelne Spielrunde beginnt durch das Auswählen der Karten durch die Spieler. `selectCards` liest die Eingaben der Spieler aus dem Eingabe-Channel aus und überprüft, ob die Eingabe valide war. Sollten alle Spieler valide Eingaben getätigt haben, wird das Einsortieren gestartet.

```
172 selectCards :: ST → Chan Msg → Chan Msg → IO ST
173 selectCards st inChan outChan = do
174   printTable outChan st
175   writeChan outChan ("", "Please␣pick␣your␣card.")
176   var ← newIORef [] — map between players and chosen cards
177   fix $ λloop → do
178     ps ← readIORef var
179     unless (length ps == length (players st)) $ do
180       (n,msg) ← readChan inChan
181       msg' ← try $ readIO msg :: IO (Either SomeException Int)
182       case msg' of
183         Right choice → do
184           let [p] = filter (==Player n undefined undefined) $ players st
185           if p `elem` map fst ps then
186             writeChan outChan (n, "You␣picked␣your␣card␣already.")
187           else if choice `notElem` hand p then
188             writeChan outChan (n, "You␣do␣not␣possess␣this␣card.␣Try␣again")
189           else do
190              writeIORef var $ (p {hand = delete choice (hand p) }, choice) : ps
191              writeChan outChan (n, "You␣chose␣" ++ show choice ++ ".␣Waiting␣for␣other␣players␣now.")
192         Left _      → writeChan outChan (n, "Cannot␣read␣your␣input.␣Try␣again.")
193       loop
194   choices ← readIORef var
195   newState ← sortIn st inChan outChan choices
196   writeChan outChan ("", "=========================================")
197   writeChan outChan ("","␣")
198   return newState
```

Das Einsortieren wird dann in der (etwas langen) Funktion `sortIn` realisiert.

```
200 sortIn :: ST → Chan Msg → Chan Msg → [(Player, Int)] → IO ST
201 sortIn st inChan outChan chosenCards = do
202   let cardSeq = sortBy (compare `on` snd) chosenCards
203   varST ← newIORef st
204   varP ← newIORef []
205   forM_ cardSeq $ \t@(p,card) → do
206     st' ← readIORef varST
207     writeIORef varST $ st' {table = Table $ sortBy (compare `on` last) (rows ∘ table $ st')}
208     st' ← readIORef varST
209     let minColumn = head ∘ map last ∘ rows ∘ table $ st'
210     if card < minColumn
211     then do
212       newState ← selectRow st' inChan outChan t
213       let [p'] = filter (==p) $ players newState
214       ps ← readIORef varP
215       writeIORef varP $ p' : ps
216       writeIORef varST newState
217     else do
218       let newRows = insertIntoRows (reverse ∘ rows ∘ table $ st') card
```

```
219       let (bigRow, rest) = partition ((>5) ∘ length) newRows
220       ps ← readIORef varP
221       if null bigRow — no row is full after insertion
222       then do
223         writeIORef varP $ p:ps
224         writeIORef varST $ st' {table = Table newRows }
225       else do
226         writeIORef varP $ p { score = score p + cardsToScore (init $ head bigRow) } : ps
227         writeIORef varST $ st' {table = Table $ [last ∘ head $ bigRow] : rest }
228   newPlayers ← readIORef varP
229   st' ← readIORef varST
230   writeIORef varST $ st' {players = newPlayers}
231   newState ← readIORef varST
232   printTable outChan newState
233   return newState
234
235 insertIntoRows :: [[Int]] → Int → [[Int]]
236 insertIntoRows [] _ = []
237 insertIntoRows (x:xs) e
238   | last x < e = (x ++ [e]) : xs
239   | otherwise  = x : insertIntoRows xs e
240
241 cardsToScore :: [Int] → Int
242 cardsToScore = sum ∘ map cardToScore
243
244 cardToScore :: Int → Int
245 cardToScore x = fiveOrTen + doublet where
246   doublet = let x' = show x
247     in if all (==head x') x' && length x' > 1 then 5 else 0
248   fiveOrTen
249     | x `mod` 10 == 0 = 3
250     | x `mod` 5 == 0 = 2
251     | otherwise = 0
```

Sollte eine Karte so niedrig sein, dass sie nirgends auf dem Tisch einsortiert werden kann, wird der User in durch die Funktion `selectRow` befragt, welche Reihe der User auf seinen Hornochsenstapel nehmen möchte.

```
253 selectRow :: ST → Chan Msg → Chan Msg → (Player, Int) → IO ST
254 selectRow st inChan outChan (p,card) = do
255   writeChan outChan (name p, "The␣card␣you␣played␣does␣not␣fit␣in␣any␣row.␣Please␣select␣a␣row␣(1–4
256   printTableForPlayer outChan (table st) p
257   fix $ λloop → do
258     (n,msg) ← readChan inChan
259     if n == name p then do
260       msg' ← try $ readIO msg :: IO (Either SomeException Int)
261       case msg' of
262         Right choice →
263           if choice `elem` [1..4]
264           then do
265             let (front, back) = splitAt (choice−1) (rows ∘ table $ st)
266             let selectedRow = head back
267             let newPlayers = p {score = score p + cardsToScore selectedRow} : filter (≠p) (players
268             let newRows = [card] : front ++ tail back
269             return $ st {players = newPlayers, table = Table newRows}
270           else do
271             writeChan outChan (name p, "This␣row␣does␣not␣exist.␣Try␣again.")
272             loop
```

```
273         Left _          → do
274           writeChan outChan (name p, "Cannot read your input. Try again.")
275           loop
276     else
277       loop
```

Das Programm wurde funktional getestet, indem mehrere Partien gespielt wurden.