Tutor: Martin Ring
Gruppe: 1
Tobias Brandt
Stefan Heitmann

Übungsblatt 3

Aufgabenlösung
Abgabe: 25.05.2017

3.1 *Aktoren in Haskell*

Punkt 1
Um diesen Punkt umzusetzen, haben wir uns dazu entschieden, den Datentyp so zu erweitern, dass die Wurzel eines Aktorensystem durch einen besonderen Konstruktor gekennzeichnet wird.

Punkt 2
Dieser Punkt ist bereits in dem vorgegebenen Framework umgesetzt.

Punkt 3
Dieser Punkt ist ebenfalls bereits in dem vorgebenen Framework umgesetzt, da die MVars die wartenden Threads automatisch in FIFO-Reihenfolge aufwecken.

Punkt 4
Diese Punkte sind ebenfalls in dem Framework umgesetzt.

Punkt 5
Dieser Punkt wurde implementiert, indem das Herein- und Herausnehmen, der Nachrichten in die MVars in einem neuen Thread gestartet wird.

Punkt 6
Dieser Punkt wurde implementiert, indem der Message-Datentyp um eine Fehler-Nachricht erweitert wurde. Zusätzlich wird in der Verhaltsschleife überprüft, ob es beim Auswerten des Verhaltens zu einem Fehler gekommen ist. Sollte dies der Fall sein, wird eine Nachricht an den Erzeuger des Aktors gesendet.

Quelltext

```haskell
module Actors (ActorRef(..), ActorContext, Behavior (..), self,
  sender, actor, ask, send, root, respond, stop, failed, become) where

import System.IO
import Control.Exception (try, evaluate, SomeException)
import Control.Concurrent

data Message a = Failed String | Stop | Message (ActorRef a) a

data ActorRef a = ActorSystem (ActorRef a) | ActorRef (ActorRef a) (MVar (Message a))

instance Eq (ActorRef a) where
  ActorRef _ var == ActorRef _ var' = var == var'
  ActorSystem a == ActorSystem b = a == b
  _ == _ = False

data ActorContext a = ActorContext {
  self :: ActorRef a,
  sender :: ActorRef a
}

newtype Behavior a = Behavior {
  receive :: ActorContext a → a → IO (Behavior a)
```

```
24 }
25
26 inbox :: ActorRef a → MVar (Message a)
27 inbox (ActorRef _ var) = var
28 inbox (ActorSystem ref) = inbox ref
29
30 root :: ActorRef a → ActorRef a
31 root (ActorRef parent _) = root parent
32 root r = r
33
34 respond :: ActorContext a → a → IO ()
35 respond context = send (sender context) (self context)
36
37 send :: ActorRef a → ActorRef a → a → IO ()
38 send recipient sender message = do
39   let recipient' = inbox recipient
40   forkIO $ putMVar recipient' (Message sender message)
41   return ()
42
43 ask :: ActorRef a → a → IO a
44 ask recipient message = do
45   inbox ← newEmptyMVar
46   let self = ActorRef undefined inbox
47   send recipient self message
48   (Message sender answer) ← takeMVar inbox
49   return answer
50
51 stop :: ActorRef a → IO ()
52 stop recipient = do
53   let recipient' = inbox recipient
54   forkIO $ putMVar recipient' Stop
55   return ()
56
57 failed :: ActorRef a → Message a → IO ()
58 failed recipient msg = do
59   let recipient' = inbox recipient
60   forkIO $ putMVar recipient' msg
61   return ()
62
63 become :: (ActorContext a → a → IO (Behavior a)) → IO (Behavior a)
64 become = return ∘ Behavior
65
66 actor :: Maybe (ActorRef a) → Behavior a → IO (ActorRef a)
67 actor parent behavior = do
68   inbox ← newEmptyMVar
69   let self = case parent of
70         Nothing → ActorSystem $ ActorRef undefined inbox
71         Just p  → ActorRef p inbox
72   let loop (Behavior behavior) = do
73         msg ← takeMVar inbox
74         case msg of
75           Stop → return ()
76           Message sender msg → do
77             let context = ActorContext self sender
78             result ← try ∘ evaluate $ behavior context msg
79             case result of
```

```
80                Right newState → do
81                  fromIO ← newState
82                  loop fromIO
83                Left err → case parent of
84                  Just parent → failed parent $ Failed ∘ show $ (err :: SomeException)
85                  Nothing → return ()
86            Failed err → do
87              hPutStrLn stderr err
88              loop $ Behavior behavior
89    forkIO $ loop behavior
90    return self
```

3.2 *Hoogle Sheets*

Wir haben uns dazu entschieden, den Server in Haskell zu implementieren. Dazu haben wir zunächst einen Parser mit der Bibliothek *Parsec* implementiert. Dabei haben wir uns ebenfalls dazu entschieden, den Reduce-Befehl in eine Faltung von Zellreferenzen zu übersetzten.

```
1  module Parser (parseFormula, Expr (..), Op(..)) where
2
3  import Text.Parsec
4  import Text.Parsec.Expr
5  import Text.Parsec.Token
6  import Text.Parsec.Language (javaStyle)
7  import Data.List
8
9  data Expr = Cell String |
10   Const Integer |
11   Binary Op Expr Expr
12   deriving Show
13
14 data Op = Plus |
15   Minus |
16   Multiply |
17   Division
18   deriving Show
19
20 lexer = makeTokenParser javaStyle
21
22 expr = buildExpressionParser table term
23
24 term =  parens lexer expr
25   <|> (Const <$> natural lexer)
26   <|> reduce
27   <|> cell
28
29 cell = do
30   col ← oneOf ['A'..'Z']
31   row ← natural lexer
32   return ∘ Cell $ col : show row
33
34 reduce = do
35   string "REDUCE"
36   parens lexer $ do
37     Cell start ← cell
38     colon lexer
39     Cell end ← cell
```

```
40      comma lexer
41      op ← operand
42      let start_col = head start
43      let end_col = head end
44      let start_row = read ∘ tail $ start :: Integer
45      let end_row = read. tail $ end :: Integer
46      let range = do
47            c ← [start_col..end_col]
48            r ← [start_row..end_row]
49            return ∘ Cell $ c:(show r)
50      return $ foldl' (Binary op) (head range) (tail range)
51
52  operand = (reservedOp lexer "*" ≫ return Multiply)
53    <|> (reservedOp lexer "/" ≫ return Division)
54    <|> (reservedOp lexer "+" ≫ return Plus)
55    <|> (reservedOp lexer "-" ≫ return Minus)
56
57
58  table = [ [Infix (reservedOp lexer "*" ≫ return (Binary Multiply)) AssocLeft,
59      Infix (reservedOp lexer "/" ≫ return (Binary Division)) AssocLeft]
60      ,[Infix (reservedOp lexer "+" ≫ return (Binary Plus))  AssocLeft,
61      Infix (reservedOp lexer "-" ≫ return (Binary Minus)) AssocLeft] ]
62
63  parseFormula :: String → Either ParseError Expr
64  parseFormula = parse expr ""
```

Da unsere Implementierung hauptsächlich aus Aktoren besteht, mussten zunächst alle Nachrichten für die Aktoren definiert werden.

```
48  data SheetMessage = CreateCell String |
49    NewCell (ActorRef SheetMessage) |
50    ParseCell String | Evaluate | ReqSubs |
51    Subs [ActorRef SheetMessage] |
52    Result Integer | Build | ReqLookUp String |
53    ResultLookUp (Maybe (ActorRef SheetMessage)) |
54    Subscribe (ActorRef SheetMessage) | Subscribed SheetMessage |
55    Failed String | Done | Update SheetMessage |
56    Terminate (ActorRef SheetMessage)| Unsubscribe
```

Zunächst haben wir den Spreadsheet-Aktor implementiert. Dieser hält eine Abbildung von alle bestehenden Zellaktoren. Der Aktor kann bei Nachrichtenerhalt einen neuen Zellaktor mit den mitgesendeten Koordinaten erstellen. Sollte an dieser Position bereits eine Zelle existieren, werden die Abonennten dieser Zelle über die neue Zelle informiert und die neue Zelle wird mit der alten Zelle ersetzt.
Ebenso stellt der Aktor eine Funktion zum Nachschlagen von Zellaktoren bereit.

```
58  spreadsheet :: (Message → IO ()) → IO (ActorRef SheetMessage)
59  spreadsheet action = actor Nothing (Behavior $ receive Map.empty) where
60    receive cells context = λcase
61      CreateCell cord → do
62        let oldCell = Map.lookup cord cells
63        subs ← do
64          case oldCell of
65            Just oc → do
66              Subs subList ← ask oc ReqSubs
67              ask oc $ Terminate oc
68              return subList
69            Nothing → return []
70        newCell ← cell (self context) cord subs action
```

```
71        respond context $ NewCell newCell
72        become $ receive $ Map.insert cord newCell cells
73      ReqLookUp cord → do
74        respond context $ ResultLookUp $ Map.lookup cord cells
75        become $ receive cells
```

Als nächstes wurde der Zellaktor implementiert. Dieser stellt eine Zelle im Spreadsheet dar. Bei Erhalt einer "Parse"-Nachricht wird die Expression der Zelle parsiert und für jeden Knoten im AST ein neuer Aktor erzeugt. Zusätzlich wird überprüft, ob der AST eine Selbstreferenz enthält. Sollte dies der Fall sein, wird die Zelle als Fehlerhaft dargestellt.
Ebenso wird bei Erhalt einer "Evaluate"-Nachricht der entsprechende Ausdruck der Zelle ausgewertet und das Ergebnis an den Client gesendet. Außerdem wartet die Zelle auf Nachrichten, die das Abonieren und Deabonieren von Zellreferenzen handhaben.

```
77  cell :: ActorRef SheetMessage → String → [ActorRef SheetMessage] → (Message → IO ()) → IO (ActorRef
78  cell parent cord subs action = actor (Just parent) (Behavior $ initial cord subs action) where
79    initial cord subs action context = λcase
80      ParseCell input → do
81        let ast' = parseFormula input
82        case ast' of
83          Right ast → do
84            if not (isLoop (Cell cord) ast) then do
85              newExpr ← expr (self context) ast
86              ask newExpr Build
87              respond context Done
88              become $ evaluating cord newExpr subs action
89            else do
90              newEmptyExpr ← empty $ self context
91              ask newEmptyExpr Build
92              respond context $ Failed "contains self reference"
93              become $ evaluating cord newEmptyExpr subs action
94          Left err → do
95            newEmptyExpr ← empty $ self context
96            ask newEmptyExpr Build
97            respond context $ Failed $ show err
98            become $ evaluating cord newEmptyExpr subs action
99    evaluating cord exprA subs action context = λcase
100     ReqSubs → do
101       respond context $ Subs subs
102       become $ evaluating cord exprA subs action
103     Evaluate → do
104       result ← ask exprA Evaluate
105       forM_ subs $ λsub → send sub (self context) $ Update result
106       respond context result
107       let x = head cord
108       let y = read ∘ tail $ cord
109       case result of
110         Result val → action $ CellUpdate x y $ Right $ Just val
111         Failed err → action $ CellUpdate x y $ Left err
112       become $ evaluating cord exprA subs action
113     Subscribe newRef → do
114       outcome ← ask exprA Evaluate
115       respond context $ Subscribed outcome
116       become $ evaluating cord exprA (newRef:subs) action
117     Unsubscribe → do
118       let subs' = delete (sender context) subs
119       become $ evaluating cord exprA subs' action
```

```
    Update _ → do
      send (self context) (self context) Evaluate
      become $ evaluating cord exprA subs action
    Terminate oc → do
      ask exprA $ Terminate oc
      stop $ self context
      respond context Done
      become undefined
    Result _ → become $ evaluating cord exprA subs action
    Failed _ → become $ evaluating cord exprA subs action
```

Die AST-Aktoren stellen Knoten im AST dar. Bei Erhalt von "Build"-Nachrichten erzeugen die Aktoren eventuell für ihre Auswertung notwendige weitere AST-Aktoren. Bei Erhalt einer "Evaluate"-Nachricht wird dann der Unterausdruck ausgewertet und das Ergebnis an den Erzeuger des Aktors weitergeleitet.

```
expr :: ActorRef SheetMessage → Expr → IO (ActorRef SheetMessage)
expr parent ast = actor (Just parent) (Behavior $ initial ast) where
  initial ast context = λcase
    Build →
      case ast of
        Cell cord → do
          cell ← cellRef (self context) cord
          ask cell Build
          respond context Done
          become $ evaluating cell
        Const val → do
          const ← const (self context) val
          ask const Build
          respond context Done
          become $ evaluating const
        Binary op e1 e2 → do
          binop ← binary (self context) e1 e2 op
          ask binop Build
          respond context Done
          become $ evaluating binop
  evaluating actor context = λcase
    Evaluate → do
      result ← ask actor Evaluate
      respond context result
      become $ evaluating actor
    Update _ → do
      send parent (self context) $ Update undefined
      become $ evaluating actor
    Terminate oc → do
      ask actor $ Terminate oc
      stop $ self context
      respond context Done
      become undefined

cellRef :: ActorRef SheetMessage → String → IO (ActorRef SheetMessage)
cellRef parent cord = actor (Just parent) (Behavior $ initial cord)  where
  initial cord context = λcase
    Build → do
      ResultLookUp result ← ask (root parent) $ ReqLookUp cord
      case result of
        Just cellA → do
          Subscribed curVal ← ask cellA $ Subscribe $ self context
```

```
173           respond context Done
174           become $ evaluating cord curVal
175         _ → do
176           NewCell emptyCell ← ask (root parent) $ CreateCell cord
177           ask emptyCell $ ParseCell ""
178           ask emptyCell $ Subscribe $ self context
179           respond context $ Failed $ "no value for: " ++ cord
180           become $ evaluating cord $ Failed $ "no value for: " ++ cord
181   evaluating cord curVal context = λcase
182     Evaluate → do
183       respond context curVal
184       become $ evaluating cord curVal
185     Update newVal → do
186       send parent (self context) $ Update undefined
187       become $ evaluating cord newVal
188     Terminate oc → do
189       send oc (self context) Unsubscribe
190       stop $ self context
191       respond context Done
192       become undefined
193
194 const :: ActorRef SheetMessage → Integer → IO (ActorRef SheetMessage)
195 const parent val = actor (Just parent) (Behavior $ initial val) where
196   initial val context = λcase
197     Build → do
198       respond context Done
199       become $ evaluating val
200   evaluating val context = λcase
201     Evaluate → do
202       respond context $ Result val
203       become $ evaluating val
204     Terminate _ → do
205       stop $ self context
206       respond context Done
207       become undefined
208
209 empty :: ActorRef SheetMessage → IO (ActorRef SheetMessage)
210 empty parent = actor (Just parent) (Behavior initial) where
211   initial context = λcase
212     Build → do
213       respond context Done
214       become $ evaluating
215   evaluating context = λcase
216     Evaluate → do
217       respond context $ Failed "N/A"
218       become $ evaluating
219     Terminate _ → do
220       stop $ self context
221       respond context Done
222       become undefined
223
224 binary :: ActorRef SheetMessage → Expr → Expr → Op → IO (ActorRef SheetMessage)
225 binary parent lhs rhs op = actor (Just parent) (Behavior $ initial lhs rhs op) where
226   initial lhs rhs op context = λcase
227     Build → do
228       exp ← expr (self context) lhs
```

```
229       exp' ← expr (self context) rhs
230       ask exp Build
231       ask exp' Build
232       respond context Done
233       become $ evaluating exp exp' op
234   evaluating l r op context = λcase
235     Evaluate → do
236       lVal ← ask l Evaluate
237       rVal ← ask r Evaluate
238       case (lVal, rVal) of
239         (Result lVal, Result rVal) → do
240           outcome ← try ∘ evaluate $ (app op) lVal rVal
241           case outcome of
242             Right result → do
243               respond context $ Result result
244               become $ evaluating l r op
245             Left err → do
246               respond context $ Failed $ show (err :: SomeException)
247               become $ evaluating l r op
248         _ → do
249           respond context $ Failed "binary failed"
250           become $ evaluating l r op
251     Update _ → do
252       send parent (self context) $ Update undefined
253       become $ evaluating l r op
254     Terminate oc → do
255       ask l $ Terminate oc
256       ask r $ Terminate oc
257       stop $ self context
258       respond context Done
259       become undefined
260
261 app :: Integral a ⇒ Op → (a → a → a)
262 app Plus = (+)
263 app Minus = (−)
264 app Multiply = (∗)
265 app Division = div
266
267 isLoop :: Expr → Expr → Bool
268 isLoop (Cell x) (Cell y) = x == y
269 isLoop (Cell x) (Const _) = False
270 isLoop c@(Cell x) (Binary _ e e') = isLoop c e || isLoop c e'
271 isLoop _ _= False
```

Tests

Getestet wurden beide Aufgaben, indem das Spreadsheet funktional ausprobiert wurde. Die Tests verliefen erfolgreich.