Reaktive Programmierung SS 2017

Tutor: Martin Ring
Gruppe: 1
Tobias Brandt
Stefan Heitmann

Übungsblatt 6

Aufgabenlösung
Abgabe: 06.07.2017

6.1 ***Hoogle Docs***

Server

messages und serialisierung

```
data ClientMessage = ClientEdit Int TextOperation Cursor

data ServerMessage = RemoteEdit TextOperation (M.Map Int Cursor)
                   | Ack
                   | HelloGuys

instance FromJSON TextAction where
  parseJSON (Number x) = do
    n ← parseJSON (Number x)
    return $
      if n < 0 then Delete $ abs n
      else Retain n
  parseJSON (String t) = do
    c ← parseJSON (String t)
    return $ Insert c

instance FromJSON TextOperation where
  parseJSON (Array a) = do
    ops ← parseJSON (Array a)
    return $ TextOperation ops

instance FromJSON ClientMessage where
  parseJSON (Object o) = ClientEdit
    <$> o .: "rev"
    <*> o .: "op"
    <*> o .: "cursor"

instance ToJSON TextAction where
  toJSON (Retain n) = toJSON (n :: Int)
  toJSON (Delete n) = toJSON (−n :: Int)
  toJSON (Insert c) = toJSON c

instance ToJSON TextOperation where
  toJSON (TextOperation as) = toJSON $ map toJSON as

instance ToJSON ServerMessage where
  toJSON Ack = toJSON ("ack" :: String)
  toJSON HelloGuys = toJSON ("helloguys" :: String)
  toJSON (RemoteEdit op cursors) = object ["op" .= toJSON op,
    "cursors" .= cursors'] where cursors' = toJSON $ M.elems cursors
```

server cursor

```
data ServerState = ServerState {
  ot :: Server TextOperation String,
  idGen :: ClientId,
  clients :: [(ClientId,WS.Connection)],
  cursors :: M.Map ClientId Cursor
}

type ClientId = Int

initialState = ServerState {
  ot = (Server ("" :: [Char]) ([] :: [TextOperation])),
  idGen = 0,
  clients = [],
  cursors = M.empty
}
```

server cursor2

```
handleSocket :: WS.Connection → ClientId → Int → MVar ServerState → IO ()
handleSocket conn id offset server = do
  −− Empfange die Daten vom Client
  let disconnect err = do
        putStrLn $ "disconnected:␣␣client␣" ⧺ show id
        modifyMVar_ server $ λs →
          return s { clients = filter ((≠ id). fst) (clients s),
            cursors = M.delete id $ cursors s }
        throw (err :: WS.ConnectionException)
  msg ← catch (WS.receiveData conn) disconnect
  case decode msg of
    Just (ClientEdit rev op cursor) → do
      print op
      op' ← modifyMVar server $ λs → do
        let (op',ot') = appendOperation (ot s) op (offset + rev)
        let newCursor = foldl (flip transformCursor) cursor $ drop (offset + rev) $ history ∘ ot $ s
        let newOtherCursors = M.map (transformCursor op) $ cursors s
        let newCursors = M.insert id newCursor newOtherCursors
```

```
          print newCursors
          forM_ (clients s) $ λ(cid,client) → do
            let cursors' = M.filterWithKey (λk _ → cid≠k) newCursors
            if cid ≠ id
            then do
              WS.sendTextData client (encode (RemoteEdit op' cursors'))
              print cursors'
            else WS.sendTextData client (encode Ack)
          return (s { ot = ot', cursors = newCursors },op')
        handleSocket conn id offset server
      Nothing → do
        print msg
        error "could␣not␣decode␣client␣message"
```

server cursor3

```
socket server pending = do
  conn ← WS.acceptRequest pending
  WS.forkPingThread conn 30
  (id,offset) ← modifyMVar server $ λs → do
    let id = 1 + idGen s
    let clients' = (id,conn) : clients s
    let offset = serverRevision (ot s) − 1
    let stateOp = serverStateOp (ot s)
    let cursors' = M.insert id 0 $ cursors s
    WS.sendTextData conn (encode (RemoteEdit stateOp $ cursors s))
    forM_ (clients s) $ λ(_, con) →
      WS.sendTextData con $ encode HelloGuys
    return (s {idGen = id, clients = clients', cursors = cursors'}, (id,offset))
  putStrLn $ "connected:␣client␣" ++ show id
  handleSocket conn id offset server
```

ot

```
type Cursor = Int

transformCursor :: TextOperation → Cursor → Cursor
transformCursor (TextOperation op) c = foldr transf c $ split op c where
  transf op c = case op of
    Retain n → c
    Insert str → c + length str
    Delete n → c − n
  split [] _ = []
  split (x:xs) c
    | c ≤ 0 = []
    | otherwise = case x of
        Retain n →
          if n ≤ c
          then x : split xs (c−n)
          else [Retain c]
        Delete n →
          if n ≤ c
          then x : split xs (c−n)
          else [Delete c]
        Insert cs →
          if length cs ≤ c
          then x : split xs (c − length cs)
          else [Insert $ take c cs]
```

ot2

```
data TextAction = Retain Int
                | Insert String
                | Delete Int deriving Show
```

ot3

```
instance Document String TextOperation where
  noop d
    | length d == 0 = TextOperation []
    | otherwise = TextOperation [Retain $ length d]
  applyOp d (TextOperation as) = apply' as d where
    apply' [] [] = []
    apply' (Retain 0 : as) cs = apply' as cs
    apply' (Retain n : as) (c:cs) = c : apply' (Retain (n−1) : as) cs
    apply' (Insert cs : as) d = cs ++ apply' as d
    apply' (Delete 0 : as) cs = apply' as cs
    apply' (Delete n : as) (_:cs) = apply' (Delete (n−1) : as) cs
```

ot4

```
instance Operation TextOperation where
  compose (TextOperation a) (TextOperation b) = TextOperation $ compose' a b where
    compose' [] [] = []
    compose' (Delete n: as) bs   = Delete n : compose' as bs
    compose' as (Insert cs : bs) = Insert cs : compose' as bs
    compose' (Retain n : as) (Retain n' : bs)
      | n > n'  = Retain n' : compose' (Retain (n−n') : as) bs
      | n < n'  = Retain n : compose' as (Retain (n'−n) : bs)
      | otherwise = Retain n : compose' as bs
    compose' (Retain n : as) (Delete n' : bs)
      | n > n'  = Delete n' : compose' (Retain (n−n') : as) bs
      | n < n'  = Delete n : compose' as (Retain (n'−n) : bs)
      | otherwise = Delete n : compose' as bs
    compose' (Insert cs : as) (Retain n : bs)
      | n > length cs = Insert cs : compose' as (Retain (n−length cs) : bs)
      | n < length cs = Insert (take n cs) : compose' (Insert (drop n cs) : as) bs
      | otherwise     = Insert cs : compose' as bs
    compose' (Insert cs : as) (Delete n :bs)
      | n > length cs = compose' as $ Delete (n − length cs) : bs
      | n < length cs = compose' ((Insert $ drop n cs) : as) bs
      | otherwise      = compose' as bs
  transform (TextOperation a) (TextOperation b) = pair $ transform' a b where
    pair (as',bs') = (TextOperation as', TextOperation bs')
    transform' [] [] = ([],[])
    transform' (Insert cs : as) bs = case transform' as bs of (as',bs') → (Insert cs : as', Retain (length cs) : bs')
    transform' as (Insert cs : bs) = case transform' as bs of (as',bs') → (Retain (length cs) : as', Insert cs : bs')
    transform' (Retain n: as) (Retain n': bs)
      | n > n' = case transform' (Retain (n−n') : as) bs of (as',bs') → (Retain n' : as', Retain n' : bs')
      | n < n' = case transform' as (Retain (n'−n) : bs) of (as',bs') → (Retain n : as', Retain n : bs')
      | otherwise = case transform' as bs of (as',bs') → (Retain n: as', Retain n': bs')
    transform' (Delete n : as) (Delete n' : bs)
      | n > n' = transform' (Delete (n−n') : as) bs
      | n < n' = transform' as $ Delete (n'−n) : bs
      | otherwise = transform' as bs
    transform' (Retain n : as) (Delete n' : bs)
      | n > n' = case transform' (Retain (n−n') : as) bs of (as',bs') → (as', Delete n' : bs')
      | n < n' = case transform' as (Delete (n'−n) : bs) of (as',bs') → (as', Delete n : bs')
      | otherwise = case transform' as bs of (as',bs') → (as', Delete n : bs')
    transform' (Delete n : as) (Retain n' : bs)
      | n > n' = case transform' (Delete (n−n') : as) bs of (as',bs') → (Delete n' : as', bs')
      | n < n' = case transform' as (Retain (n'−n) : bs) of (as',bs') → (Delete n : as', bs')
      | otherwise = case transform' as bs of (as',bs') → (Delete n: as', bs')
```

Client

move cursor bei actions

```
  def insert(c: Char) = {
    val operation = if (cursor == 0 && content.length == 0) {
      List(Insert(c.toString()))
    } else if(cursor == 0) {
      List(Insert(c.toString()),Retain(content.length))
    } else if(content.length == 0) {
      List(Insert(c.toString()))
    } else if(content.length == cursor) {
      List(Retain(cursor),Insert(c.toString()))
    } else {
      List(Retain(cursor),Insert(c.toString()),Retain(content.length − cursor))
    }
    (Option(operation),Editor(TextOperation.applyOp(content,operation),cursor + 1,
      cursors.map(c ⇒ TextOperation.transformCursor(operation, c))))
  }

  def backspace = if (cursor > 0) {
    val operation = List(Retain(cursor − 1), Delete(1) , Retain(content.length − cursor)).filter(op ⇒ op!=Retain(0))
    (Option(operation),Editor(TextOperation.applyOp(content,operation),cursor − 1,
      cursors.map(c ⇒ TextOperation.transformCursor(operation, c))))
  } else (None,this)

  def moveLeft = {
    val operation = List(Retain(content.length))
    (Option(operation), Editor(content, Math.max(0,cursor − 1),cursors))
  }

  def moveRight = {
    val operation = List(Retain(content.length))
    (Option(operation), Editor(content, Math.min(content.length, cursor + 1),cursors))
  }
```

empfangen von edits und mehr :)

```
      val editor = Flow[ClientEvent].scan((ClientState.empty,Option.empty[ClientMessage])) {
        case ((state,lastMessage),Receive(message)) ⇒ message match {
          case Ack ⇒
            val (op, newClient) = state.ot.ack
            val message = op.map(ClientEdit(newClient.revision, _, 1))
            (state.copy(ot = newClient), message)
          case HelloGuys ⇒
            val newCursors = state.editor.cursors :+ 0
            (ClientState(state.ot, Editor(state.editor.content, state.editor.cursor, newCursors)), None)
          case RemoteEdit(op, cursors) ⇒
            val (top, newClient) = state.ot.remoteEdit(op)
            val newContent = TextOperation.applyOp(state.editor.content, top)
            val newCursorPos = TextOperation.transformCursor(op, state.editor.cursor)
            val pendingCursors = state.ot.pending match {
              case None ⇒ cursors
              case Some(o) ⇒ cursors.map(c ⇒ TextOperation.transformCursor(o,c))
            }
            val newCursors = state.ot.buffer match  {
                case None ⇒ cursors
                case Some(o) ⇒ cursors.map(c ⇒ TextOperation.transformCursor(o,c))
            }
            val newState = state.copy(
              ot = newClient,
              editor = Editor(newContent, Math.min(newContent.length, newCursorPos), newCursors)
            )
            (newState, None)
        }
        case ((state,lastMessage),Keystroke(key)) ⇒
          val (op, newEditor) = key match {
            case KeyValue.Backspace ⇒ state.editor.backspace
            case KeyValue.ArrowLeft ⇒ state.editor.moveLeft
            case KeyValue.ArrowRight ⇒ state.editor.moveRight
            case KeyValue.Enter ⇒ state.editor.insert('\n')
            case other if other.length == 1 ⇒ state.editor.insert(other.head)
            case other ⇒ (None, state.editor)
          }
          val (newClient,message) =
            op.fold(
              (state.ot,Option.empty[ClientMessage])
            ) { op ⇒
              val (synced,newState) = state.ot.localEdit(op)
              val msg = if (synced) Some(ClientEdit(newState.revision,op, newEditor.cursor)) else None
              (newState,msg)
            }
        (ClientState(newClient,newEditor),message)
    }.recover {
      case NonFatal(e) ⇒
        (ClientState(Client.empty(TextOperation), Editor(e.getMessage,0,List())),None)
    }
```

Transformcursor und apply op

```
  def transformCursor(op: Operation, cursor: Int) : Int =
    TextOperation.split(op,cursor).foldRight(cursor) {
      (o, acc) ⇒ o match {
        case Retain(n) ⇒ acc
        case Delete(n) ⇒ acc − n
        case Insert(cs) ⇒ acc + cs.length
      }
    }

  def split(op: Operation, c : Int) : Operation = {
    op match {
      case Nil ⇒ Nil
      case (x::xs) ⇒
        if (c ≤ 0) Nil
        else x match {
          case Retain(n) ⇒
            if (n ≤ c) x :: split(xs,c−n)
            else List(Retain(c))
          case Delete(n) ⇒
            if (n ≤ c) x :: split(xs,c−n)
            else List(Delete(c))
          case Insert(cs) ⇒
            if (cs.length ≤ c) x :: split(xs,c−cs.length)
            else List(Insert(cs.take(c)))
        }
    }
  }
    def applyOp(doc: Document, op: Operation): Document = (doc,op) match {
    case ("",Nil) ⇒ ""
    case (d,Retain(0)::as) ⇒ applyOp(d,as)
    case (d,Retain(n)::as) if d.nonEmpty ⇒ d.head + applyOp(d.tail,Retain(n−1)::as)
    case (d,Insert(cs)::as) ⇒ cs ++ applyOp(d,as)
    case (d,Delete(0)::as) ⇒ applyOp(d,as)
    case (d,Delete(n)::as) if d.nonEmpty ⇒ applyOp(d.tail, Delete(n−1)::as)
  }
```

rendern der cursor angepasst

```
  def render(elem: html.Div) = {
    elem.textContent = content
    val (w,h) = measure(content.take(cursor))
    val ownCursor = document.createElement("div").asInstanceOf[html.Div]
    ownCursor.style.left = w + "px"
    ownCursor.style.top = h + "px"
    ownCursor.classList.add("ownCaret")
    elem.appendChild(ownCursor)

    cursors.foldRight(elem)((c,e) ⇒ {
      val (w,h) = measure(content.take(c))
      val cursor = document.createElement("div").asInstanceOf[html.Div]
      cursor.style.left = w + "px"
      cursor.style.top = h + "px"
      cursor.classList.add("caret")
      e.appendChild(cursor)
      e
    })
  }
```

compose und transform

```
  def compose(a: Operation, b: Operation): Operation = (a,b) match {
    case (Nil, Nil)          ⇒ Nil
    case (Delete(n)::as, bs) ⇒ Delete(n) :: compose(as, bs)
    case (as, Insert(cs)::bs) ⇒
      Insert(cs) :: compose(as, bs)
    case (Retain(n)::as, Retain(n2)::bs) ⇒
      if (n > n2) Retain(n2)::compose(Retain(n−n2)::as,bs)
      else if (n < n2) Retain(n)::compose(as,Retain(n2−n)::bs)
      else Retain(n)::compose(as, bs)
    case (Retain(n):: as, Delete(n2)::bs) ⇒
      if(n > n2) Delete(n2)::compose(Retain(n−n2)::as,bs)
      else if(n < n2) Delete(n)::compose(as,Retain(n2−n)::bs)
      else Delete(n)::compose(as,bs)
    case (Insert(cs)::as,Retain(n)::bs) ⇒
      if(n > cs.length) Insert(cs)::compose(as,Retain(n−cs.length)::bs)
      else if(n < cs.length) Insert(cs.take(n))::compose(Insert(cs.drop(n))::as, bs)
      else Insert(cs) :: compose(as, bs)
    case (Insert(cs)::as, Delete(n)::bs) ⇒
      if(n > cs.length) compose(as,Delete(n−cs.length)::bs)
      else if(n < cs.length) compose(Insert(cs.drop(n))::as,bs)
      else compose(as, bs)
  }

  def transform(a: Operation, b: Operation): (Operation,Operation) = (a,b) match {
    case (Nil,Nil) ⇒ (Nil,Nil)
    case (Insert(cs)::as, bs) ⇒
      val (as_,bs_) = transform(as,bs)
      (Insert(cs)::as_,Retain(cs.length)::bs_)
    case (as, Insert(cs) :: bs) ⇒
      val (as_,bs_) = transform(as,bs)
      (Retain(cs.length)::as_, Insert(cs)::bs_)
    case (Retain(n):: as, Retain(n2):: bs) ⇒
      if (n > n2) {
        val (as_,bs_) = transform(Retain(n−n2)::as,bs)
        (Retain(n2)::as_,Retain(n2)::bs_)
      }
      else if (n < n2) {
        val (as_,bs_) = transform(as,Retain(n2−n) :: bs)
        (Retain(n) :: as_, Retain(n) :: bs_)
      }
      else {
        val (as_,bs_) = transform(as,bs)
        (Retain(n) ::  as_, Retain(n2) ::  bs_)
      }
    case (Delete(n) :: as, Delete(n2) :: bs) ⇒
      if(n > n2) {
        transform(Delete(n−n2) :: as,bs)
      }
      else if(n > n2) {
        transform(as,Delete(n2−n) :: bs)
      }
      else {
        transform(as,bs)
      }
    case (Retain(n) ::  as, Delete(n2) :: bs) ⇒
      if(n > n2) {
        val (as_,bs_) = transform(Retain(n−n2) :: as,bs)
        (as_,Delete(n2) :: bs_)
      }
      else if(n < n2) {
        val (as_,bs_) = transform(as,Delete(n2−n) :: bs)
        (as_,Delete(n) :: bs_)
      }
      else {
        val (as_,bs_) = transform(as,bs)
        (as_, Delete(n) :: bs_)
      }
    case (Delete(n) ::  as, Retain(n2) :: bs) ⇒
      if (n > n2) {
        val (as_,bs_) = transform(Delete(n−n2) :: as, bs)
        (Delete(n2) :: as_,bs_)
      }
      else if(n < n2) {
        val (as_,bs_) = transform(as,Retain(n2−n) :: bs)
        (Delete(n) :: as_,bs_)
      }
      else {
        val (as_,bs_) = transform(as,bs)
        (Delete(n) :: as_,bs_)
      }
  }
```

reduce der inserts

```
  def localEdit(op: ot.Operation): (Boolean, Client[T]) = {
    val newPending = pending orElse Some(op)
    val newBuffer = for {
      pending ← pending
    } yield buffer.map(ot.compose(_,op)).map(ot.reduce(_)).getOrElse(op)
    (pending.isEmpty, copy(pending = newPending, buffer = newBuffer))
  }
```

Tests

TODO