Reaktive Programmierung
Vorlesung 3 vom 24.04.2019
Nebenläufigkeit: Futures and Promises

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

Fahrplan

- Einführung
- Monaden und Monadentransformer
- Nebenläufigkeit: Futures and Promises
- Aktoren I: Grundlagen
- Aktoren II: Implementation
- Meta-Programmierung
- Bidirektionale Programmierung
- Reaktive Ströme I
- Reaktive Ströme II
- Funktional-Reaktive Programmierung
- Software Transactional Memory
- Eventual Consistency
- Robustheit und Entwurfsmuster
- Theorie der Nebenläufigkeit, Abschluss

Inhalt

- Konzepte der Nebenläufigkeit
- Nebenläufigkeit in Scala und Haskell
- Futures and Promises

Konzepte der Nebenläufigkeit

Begrifflichkeiten

- **Thread (lightweight process)** vs. **Prozess**

Programmiersprache/Betriebssystem	Betriebssystem
(z.B. Java, Haskell/Linux)	
gemeinsamer Speicher	getrennter Speicher
Erzeugung billig	Erzeugung teuer
mehrere pro Programm	einer pro Programm

- Multitasking:
  - **präemptiv:** Kontextwechsel wird erzwungen
  - **kooperativ:** Kontextwechsel nur freiwillig

Threads in Java

- Erweiterung der Klassen `Thread` oder `Runnable`
- Gestartet wird Methode `run()` — durch eigene überladen
- Starten des Threads durch Aufruf der Methode `start()`
- Kontextwechsel mit `yield()`
- Je nach JVM kooperativ **oder** präemptiv.
- Synchronisation mit **Monitoren** (`synchronize`)

Threads in Scala

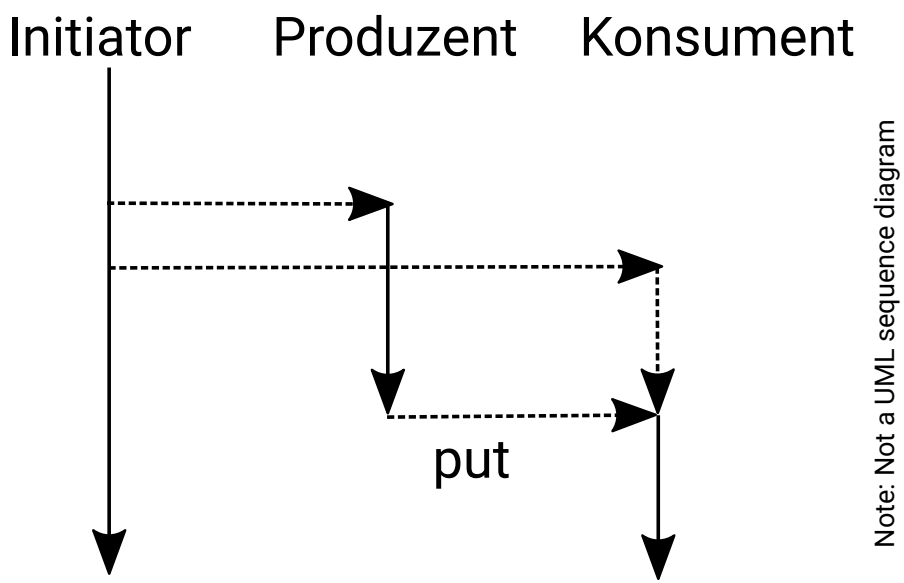
- Scala nutzt das Threadmodell der JVM
  - Kein sprachspezifisches Threadmodell
- Daher sind Threads vergleichsweise **teuer**.
- Synchronisation auf unterster Ebene durch Monitore (synchronized)
- Bevorzugtes Abstraktionsmodell: **Aktoren** (dazu später mehr)

Threads in Haskell: Concurrent Haskell

- Sequentielles Haskell: Reduktion eines Ausdrucks
  - Auswertung
- Nebenläufiges Haskell: Reduktion eines Ausdrucks an **mehreren Stellen**
  - `ghc` implementiert Haskell-Threads
  - Zeitscheiben (Default 20ms), Kontextwechsel bei Heapallokation
  - Threaderzeugung und Kontextswitch sind **billig**
- Modul `Control.Concurrent` enthält Basisfunktionen
- Wenige Basisprimitive, darauf aufbauend Abstraktionen
- Synchronisation mit Futures

Futures

- Futures machen Nebenläufigkeit **explizit**
- Grundprinzip:
  - Ausführung eines Threads wird **verzögert**
  - Konsument startet erst, wenn Ergebnis vorhanden.



Futures in Scala

Futures in Scala

- Antwort als **Callback**:

```
trait Future[+T] {
  def onComplete(f: Try[T] ⇒ Unit): Unit
  def map[U](f: T ⇒ U): Future[U]
  def flatMap[U](f: T ⇒ Future[U]): Future[U]
  def filter(p: T ⇒ Boolean): Future[T]
}
```

- `map`, `flatMap`, `filter` für monadische Notation
- Factory-Methode für einfache Erzeugung
- Vordefiniert in `scala.concurrent.Future`, Beispielimplementation `Future.scala`

Beispiel: Robot.scala

- Roboter, kann sich um n Positionen bewegen:

```
    if (n ≤ 0) this
    else if (battery > 0) {
      Thread.sleep(100*Random.nextInt(10));
      Robot(id, pos+1, battery- 1).mv(n-1)
    } else throw new LowBatteryException

  def move(n: Int): Future[Robot] = Future { mv(n) }

  override def toString = s"Robot #$id at $pos [battery:
      $battery]"
```

Beispiel: Moving the robots

```
object Examples {
  def ex1 = {
    val robotSwarm = List.range(1,6).map{i⇒ Robot(i,0,10)}
    val moved = robotSwarm.map(_.move(10))
    moved.map(_.onComplete(println))
    println("Started moving...")
```

- 6 Roboter erzeugen, alle um zehn Positionen bewegen.
- Wie lange dauert das?
  - 0 Sekunden (nach spät. 10 Sekunden Futures erfüllt)
- Was wir verschweigen: ExecutionContext

Compositional Futures

- Wir können Futures komponieren
  - "Spekulation auf die Zukunft"
- Beispiel: Roboterbewegung

```
def ex2 = { val r= Robot(99, 0, 20); for {
  r1 ← r.move(3)
  r2 ← r1.move(5)
  r3 ← r2.move(2)
```

- Fehler (Failure) werden propagiert

Promises

- Promises sind das Gegenstück zu Futures

```
trait Promise {
  def complete(result: Try[T])
  def success(result: T)
  def future: Future[T]
}

object Promise {
  def apply[T]: Promise[T] = ...
}
```

- Das Future eines Promises wird durch die complete Methode **erfüllt**.

Futures in Haskell

Concurrent Haskell: Wesentliche Typen und Funktionen

- Jeder Thread hat einen Identifier: abstrakter Typ ThreadId
- Neuen Thread erzeugen: forkIO :: IO()→ IO ThreadId
- Thread stoppen: killThread :: ThreadId → IO ()
- Kontextwechsel: yield :: IO ()
- Eigener Thread: myThreadId :: IO ThreadId
- Warten: threadDelay :: Int → IO ()

Concurrent Haskell — erste Schritte

- Ein einfaches Beispiel:

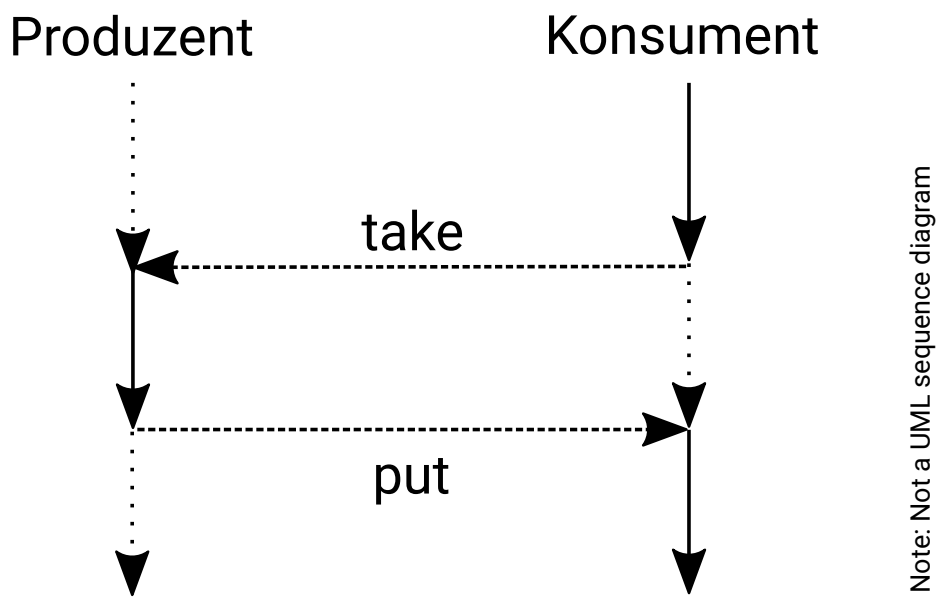
```
write :: Char→ IO ()
write c = do putChar c; write c

main :: IO ()
main = do forkIO (write 'X'); write 'O'
```

- Ausgabe `ghc`: $(\texttt{X}^*|\texttt{O}^*)^*$

Futures in Haskell: MVars

- **Basissynchronisationmechanismus** in Concurrent Haskell
  - Alles andere abgeleitet
- Grundprinzip:



Futures in Haskell: MVars

- MVar α ist **polymorph** über dem Inhalt
- Entweder **leer** oder **gefüllt** mit Wert vom Typ α
- Verhalten beim Lesen und Schreiben:

Zustand vorher:	leer	gefüllt
Lesen	blockiert (bis gefüllt)	danach leer
Schreiben	danach gefüllt	blockiert (bis leer)

  - NB. Aufwecken blockierter Prozesse einzeln in FIFO

Basisfunktionen MVars

- Neue Variable erzeugen (leer oder gefüllt):

```
newEmptyMVar :: IO (MVar α)
newMVar :: α → IO (MVar α)
```

- Lesen:

```
takeMVar :: MVar α → IO α
```

- Schreiben:

```
putMVar :: MVar α → α → IO ()
```

- Es gibt noch weitere (nicht-blockierend lesen/schreiben, Test ob gefüllt, map etc.)

Ein einfaches Beispiel: Robots Revisited

```
data Robot = Robot {id :: Int, pos :: Int, battery :: Int}
```

- Hauptfunktion: MVar anlegen, nebenläufig Bewegung starten

```
move :: Robot→ Int→ IO (MVar Robot)
move r n = do
  m ← newEmptyMVar; forkIO (mv m r n); return m
```

- Bewegungsfunktion:

```
mv :: MVar Robot → Robot→ Int→ IO ()
mv v r n
   | n ≤ 0 = putMVar v r
   | otherwise = do
     m← randomRIO(0,10); threadDelay(m∗100000)
     mv v r{pos= pos r+1, battery= battery r− 1} (n−1)
```

Abstraktion von Futures

- Aus MVar α konstruierte Abstraktionen
- Semaphoren (QSem aus Control.Concurrent.QSem):

```
waitQSem   :: QSem → IO ()
signalQSem :: QSem → IO ()
```

  - Siehe Sem.hs
  - Damit auch synchronized wie in Java (huzzah!)

- Kanäle (Chan α aus Control.Concurrent.Chan):

```
writeChan :: Chan α → α → IO ()
readChan  :: Chan α → IO α
```

Asynchrone Ausnahmen

- Ausnahmen unterbrechen den sequentiellen Kontrollfluß
- In Verbindung mit Nebenläufigkeit **überraschende Effekte**:

```
m ← newEmptyMVar
forkIO (do {s← takeMVar m; putStrLn s})
threadDelay (100000)
putMVar m (error "FOO!")
```

- In welchem Thread wird die Ausnahme geworfen?

Asynchrone Ausnahmen

- Ausnahmen unterbrechen den sequentiellen Kontrollfluß
- In Verbindung mit Nebenläufigkeit **überraschende Effekte**:

```
m ← newEmptyMVar
forkIO (do {s← takeMVar m; putStrLn s})
threadDelay (100000)
putMVar m (error "FOO!")
```

- In welchem Thread wird die Ausnahme geworfen?
- Wo kann sie gefangen werden?

Asynchrone Ausnahmen

- Ausnahmen unterbrechen den sequentiellen Kontrollfluß
- In Verbindung mit Nebenläufigkeit **überraschende Effekte**:

```
m ← newEmptyMVar
forkIO (do {s← takeMVar m; putStrLn s})
threadDelay (100000)
putMVar m (error "FOO!")
```

- In welchem Thread wird die Ausnahme geworfen?
- Wo kann sie gefangen werden?
- Deshalb haben in Scala die Future-Callbacks den Typ:

```
trait Future[+T] { def onComplete(f: Try[T] ⇒ Unit): Unit
```

Explizite Fehlerbehandlung mit Try

- Die Signatur einer Methode verrät nichts über mögliche Fehler:

```
    if (n ≤ 0) this
    else if (battery > 0) {
```

- Try[T] macht Fehler explizit (**Materialisierung** oder Reifikation):

```
sealed abstract class Try[+T] {
  def flatMap[U](f: T ⇒ Try[U]): Try[U] = this match {
    case Success(x) ⇒
      try f(x) catch { case NonFatal(ex) ⇒ Failure(ex) }
    case fail: Failure ⇒ fail  }

case class Success[T](x: T) extends Try[T]
case class Failure(ex: Throwable) extends Try[Nothing]
```

- Ist Try eine Monade?

Explizite Fehlerbehandlung mit Try

- Die Signatur einer Methode verrät nichts über mögliche Fehler:

```
    if (n ≤ 0) this
    else if (battery > 0) {
```

- Try[T] macht Fehler explizit (**Materialisierung** oder Reifikation):

```
sealed abstract class Try[+T] {
  def flatMap[U](f: T ⇒ Try[U]): Try[U] = this match {
    case Success(x) ⇒
      try f(x) catch { case NonFatal(ex) ⇒ Failure(ex) }
    case fail: Failure ⇒ fail  }

case class Success[T](x: T) extends Try[T]
case class Failure(ex: Throwable) extends Try[Nothing]
```

- Ist Try eine Monade? Nein, Try(e) flatMap f ≠ f e

Zusammenfassung

- **Nebenläufigkeit in Scala** basiert auf der JVM:
  - Relativ schwergewichtige Threads, Monitore (synchronized)
- **Nebenläufigkeit in Haskell**: Concurrent Haskell
  - Leichtgewichtige Threads, MVar
- **Futures**: Synchronisation über veränderlichen Zustand
  - In Haskell als MVar mit Aktion (IO)
  - In Scala als Future mit Callbacks
- Explizite Fehler bei Nebenläufigkeit **unverzichtbar**
- Morgen: Scala Collections, nächste VL: das Aktorenmodell