

Reaktive Programmierung  
Vorlesung 3 vom 03.05.2022  
Nebenläufigkeit: Futures and Promises

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

# Fahrplan

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

# Inhalt

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

# I. Konzepte der Nebenläufigkeit

# Begrifflichkeiten

- | ▶ <b>Thread (lightweight process)</b>                           | vs. | <b>Prozess</b>         |
|---|-----|------------------------|
| Programmiersprache/Betriebssystem<br>(z.B. Java, Haskell/Linux) |     | Betriebssystem         |
| gemeinsamer Speicher  |     | getrennter Speicher    |
| Erzeugung <b>billig</b>   |     | Erzeugung <b>teuer</b> |
| mehrere pro Programm  |     | einer pro Programm     |
- 
- ▶ Multitasking:
    - ▶ **präemptiv**: Kontextwechsel wird **erzungen**
    - ▶ **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)

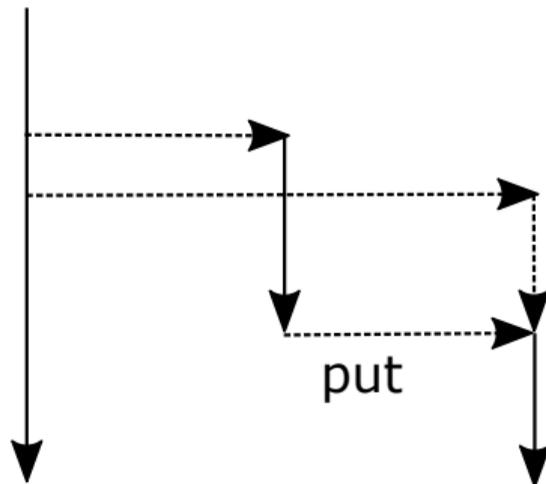
# Threads in Haskell: Concurrent Haskell

- ▶ **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.

Initiator    Produzent    Konsument



Note: Not a UML sequence diagram

## II. 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]  
  
object Future:  
  def apply[T](f: => T): 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:

```
private def mv(n: Int): Robot =  
  if n ≤ 0 then this  
  else if (battery > 0) then  
    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

```
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)  
  yield r3
```

- ▶ 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**.

# III. 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

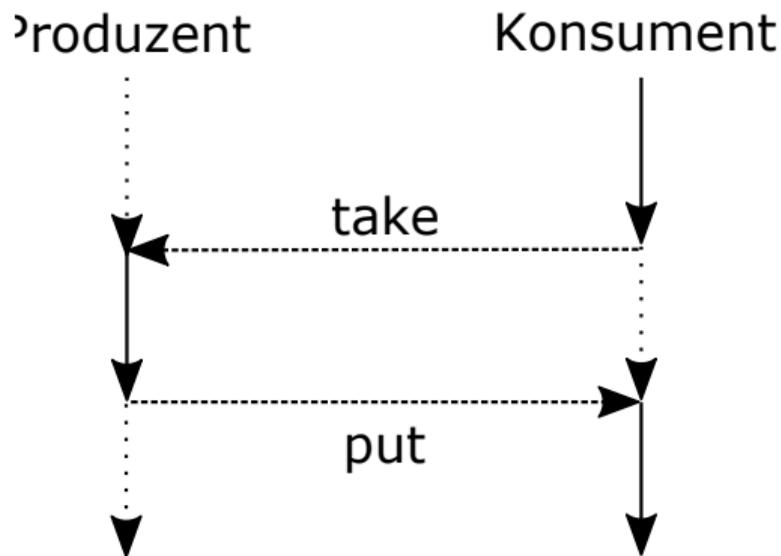
- ▶ Ein einfaches Beispiel:

```
write :: Char → IO ()  
write c = do putChar c; write c  
  
main :: IO ()  
main = do forkIO (write 'X'); write '0'
```

- ▶ Ausgabe ghc:  $(X^*|0^*)^*$

# Futures in Haskell: MVars

- ▶ **Basissynchronisationsmechanismus** in Concurrent Haskell
  - ▶ Alles andere abgeleitet
- ▶ Grundprinzip:



note: Not a UML sequence diagram

# Futures in Haskell: MVars

- ▶ `MVar  $\alpha$`  ist **polymorph** über dem Inhalt
- ▶ Entweder **leer** oder **gefüllt** mit Wert vom Typ  $\alpha$
- ▶ Verhalten beim Lesen und Schreiben:

Zustand vorher:	leer	gefüllt
Lesen	<b>blockiert</b> (bis gefüllt)	danach leer
Schreiben	danach gefüllt	<b>blockiert</b> (bis leer)

- ▶ NB. **Aufwecken** blockierter Prozesse **einzeln** in **FIFO**

# Basisfunktionen MVars

- ▶ Neue Variable erzeugen (leer oder gefüllt):

```
newEmptyMVar :: IO (MVar  $\alpha$ )  
newMVar     ::  $\alpha \rightarrow$  IO (MVar  $\alpha$ )
```

- ▶ Lesen:

```
takeMVar :: MVar  $\alpha \rightarrow$  IO  $\alpha$ 
```

- ▶ Schreiben:

```
putMVar :: MVar  $\alpha \rightarrow \alpha \rightarrow$  IO ()
```

- ▶ Es gibt noch weitere (nicht-blockierend lesen/schreiben, Test ob gefüllt, 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
  f ← newEmptyMVar; forkIO (mv f r n); return f where
```

- ▶ Bewegungsfunktion (lokal zu `move`):

```
mv f r n
  | n ≤ 0 || battery r ≤ 0 = putMVar f r
  | otherwise = do
    m ← randomRIO(0,10); threadDelay(m*100000)
    putStrLn $ "Bleep, bleep: " ++ show r
    mv f 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)
catcherr $ 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)
catcherr $ 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)
catcherr $ 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:

```
private def mv(n: Int): Robot =
```

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

```
enum Try[+T]:
```

```
  case Success(x: T)
```

```
  case Failure(ex: Throwable)
```

```
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
```

- ▶ Ist Try eine Monade?

## Explizite Fehlerbehandlung mit Try

- ▶ Die Signatur einer Methode verrät nichts über mögliche Fehler:

```
private def mv(n: Int): Robot =
```

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

```
enum Try[+T]:
```

```
  case Success(x: T)
```

```
  case Failure(ex: Throwable)
```

```
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
```

- ▶ Ist Try eine Monade? Nein,  $\text{Try}(e) \text{ flatMap } f \neq f e$

# 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**
- ▶ Nächste VL: das Aktorenmodell