

# Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

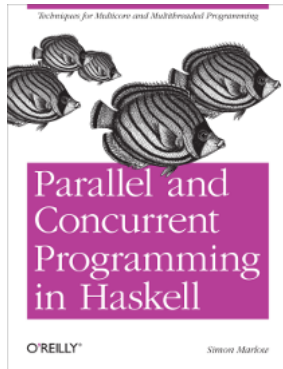
# Outline I

Übersicht für Heute:

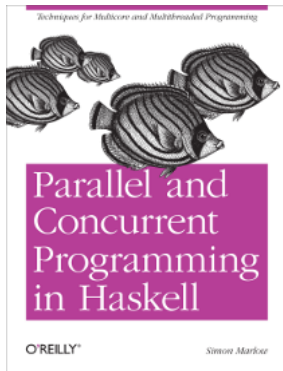
- 1 Wiederholung
- 2 Threads, MVars, etc.
- 3 Software Transactional Memory
- 4 Parallelism through concurrency
- 5 Distributed Programming

# Wiederholung

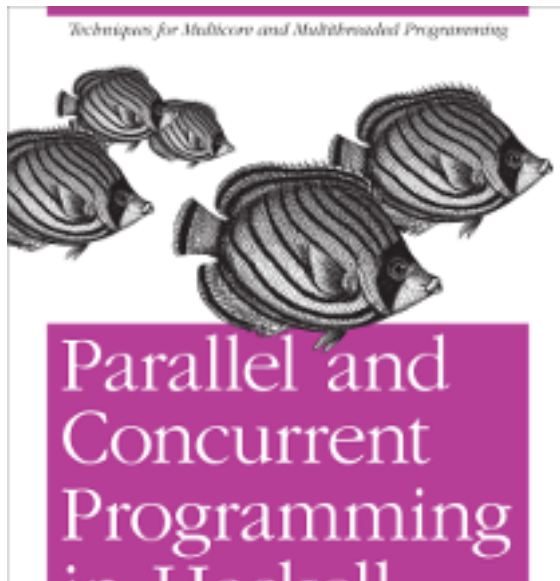
## Leseempfehlung:



## Leseempfehlung:



... srsly!



## Überblick:

## Überblick:

### Parallelism:

- Mehrere Hardwareelemente



## Überblick:

### Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen

## Überblick:

### Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)

## Überblick:

### Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

## Überblick:

### Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

### Concurrency:

- Mehrere Threads

## Überblick:

### Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

### Concurrency:

- Mehrere Threads
- Dinge gleichzeitig tun

## Überblick:

### Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

### Concurrency:

- Mehrere Threads
- Dinge gleichzeitig tun
- nichtdeterministisch

## Überblick:

### Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

### Concurrency:

- Mehrere Threads
- Dinge gleichzeitig tun
- nichtdeterministisch
- oft impertativ

# Die Basics: Threads, MVars, etc.



Wir beginnen mit der Funktion, die einen neuen Thread erstellt:

```
forkIO :: IO () -> IO ThreadId
```

Wir beginnen mit der Funktion, die einen neuen Thread erstellt:

```
forkIO :: IO () -> IO ThreadId
```

Threads interagieren notwendigerweise mit der Welt, ergo ist die Berechnung, die wir übergeben vom Typ `IO ()`.

Wir beginnen mit der Funktion, die einen neuen Thread erstellt:

```
forkIO :: IO () -> IO ThreadId
```

Threads interagieren notwendigerweise mit der Welt, ergo ist die Berechnung, die wir übergeben vom Typ `IO ()`.

Die `ThreadId` kann später benutzt werden um z.B. den Thread vorzeitig zu töten oder ihm eine Exception zuzuschmeißen.

Ein kleines Beispiel:

```
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO ()
main = do
    hSetBuffering stdout NoBuffering
    forkIO (replicateM_ 100000 (putChar 'A'))
    replicateM_ 100000 (putChar 'B')
```

Ein kleines Beispiel:

```
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO ()
main = do
    hSetBuffering stdout NoBuffering
    forkIO (replicateM_ 100000 (putChar 'A'))
    replicateM_ 100000 (putChar 'B')
```

...Output?

Ein kleines Beispiel:

```
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO ()
main = do
    hSetBuffering stdout NoBuffering
    forkIO (replicateM_ 100000 (putChar 'A'))
    replicateM_ 100000 (putChar 'B')
```

...Output?

```
AAAAAAAAABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABAB
```

Aber...

Aber... wie kriegen wir jetzt Ergebnisse aus der Berechnung raus?  
Der Typ ist nur `IO ()`, das liefert nichts (interessantes) zurück!



Aber... wie kriegen wir jetzt Ergebnisse aus der Berechnung raus?  
Der Typ ist nur `IO ()`, das liefert nichts (interessantes) zurück!

Das gleiche Problem hatten wir schon in der `Par`-Monade. Lösung damals waren `IVars`:

```
data IVar a  -- instance Eq

new :: Par (IVar a)
put  :: NFData a => IVar a -> a -> Par ()
get  :: IVar a -> Par a
```

Introducing: ...

## Introducing: ...MVars!

```
data MVar a  -- abstract

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()

readMVar     :: MVar a -> IO a
```

## Introducing: ...MVars!

```
data MVar a -- abstract

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()

readMVar     :: MVar a -> IO a
```

Wir brauchen hier keine eigene Monade wie `Par`. Da `Concurrency` so oder so effektiv ist, reicht `IO` vollkommen aus.

Unterschied zwischen `IVars` und `MVars`: erstere sind *immutable*, letztere sind *mutable*.

Ein Beispiel zu MVars:

```
main :: IO ()
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r <- takeMVar m
  print r
  r <- takeMVar m
  print r
```

Ein Beispiel zu MVars:

```
main :: IO ()
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r <- takeMVar m
  print r
  r <- takeMVar m
  print r
```

Wie wir sehen kann die gleiche MVar über Zeit mehrere Zustände annehmen und erfolgreich zur Kommunikation zwischen Threads benutzt werden.

Generell haben MVars drei Hauptaufgaben:

Generell haben MVars drei Hauptaufgaben:

- **Channel mit nur einem Slot**

Eine MVar kann als Nachrichtenkanal zwischen Threads benutzt werden, allerdings maximal eine Nachricht auf einmal halten.



Generell haben MVars drei Hauptaufgaben:

- **Channel mit nur einem Slot**

Eine MVar kann als Nachrichtenkanal zwischen Threads benutzt werden, allerdings maximal eine Nachricht auf einmal halten.

- **Behälter für shared mutable state**

In Concurrent Haskell brauchen oft mehrere Threads Zugriff auf einen shared state. Ein beliebtes Designpattern ist, das dieser State als normaler (immutable) Haskell-Datentyp repräsentiert und in einer MVar verpackt wird.

Generell haben MVars drei Hauptaufgaben:

- **Channel mit nur einem Slot**

Eine MVar kann als Nachrichtenkanal zwischen Threads benutzt werden, allerdings maximal eine Nachricht auf einmal halten.

- **Behälter für shared mutable state**

In Concurrent Haskell brauchen oft mehrere Threads Zugriff auf einen shared state. Ein beliebtes Designpattern ist, das dieser State als normaler (immutable) Haskell-Datentyp repräsentiert und in einer MVar verpackt wird.

- **Baustein für kompliziertere Strukturen**

# Software Transactional Memory (STM)

# Parallelism through Concurrency

# Distributed Programming