Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

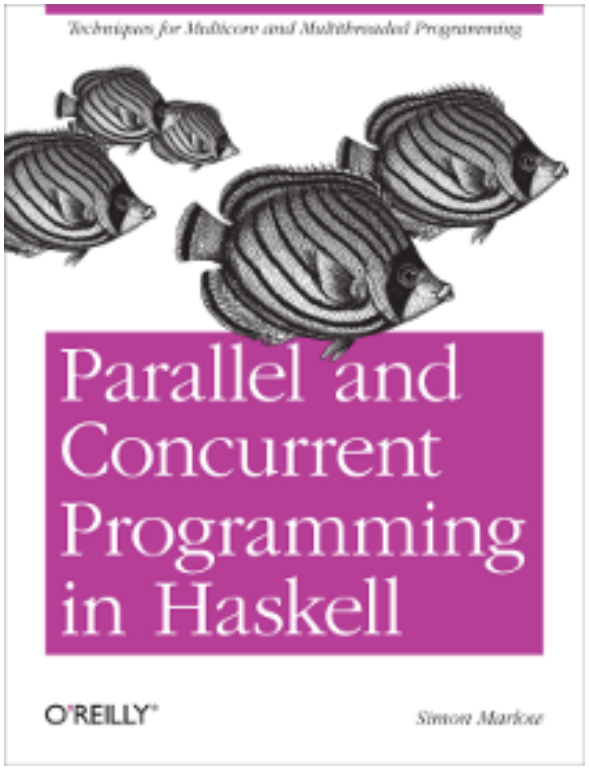
Jonas Betzendahl & Stefan Dresselhaus

Outline I

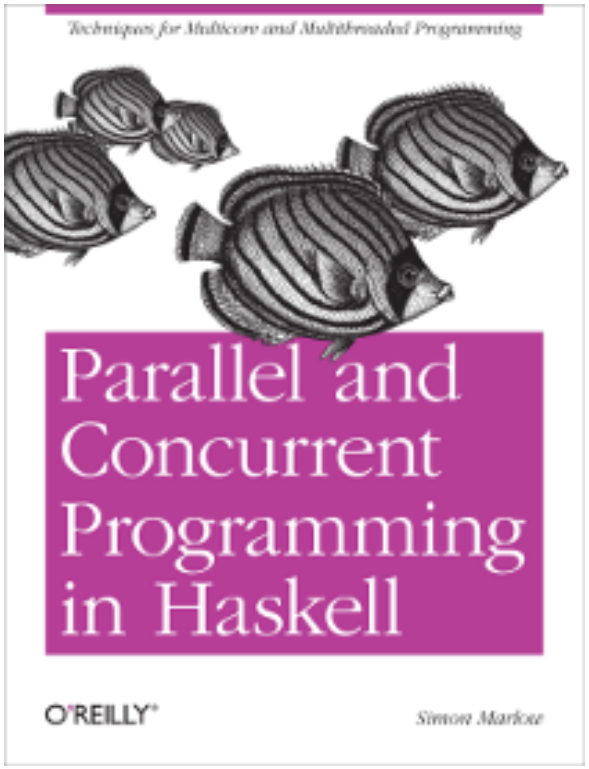
Übersicht für Heute:

Wiederholung

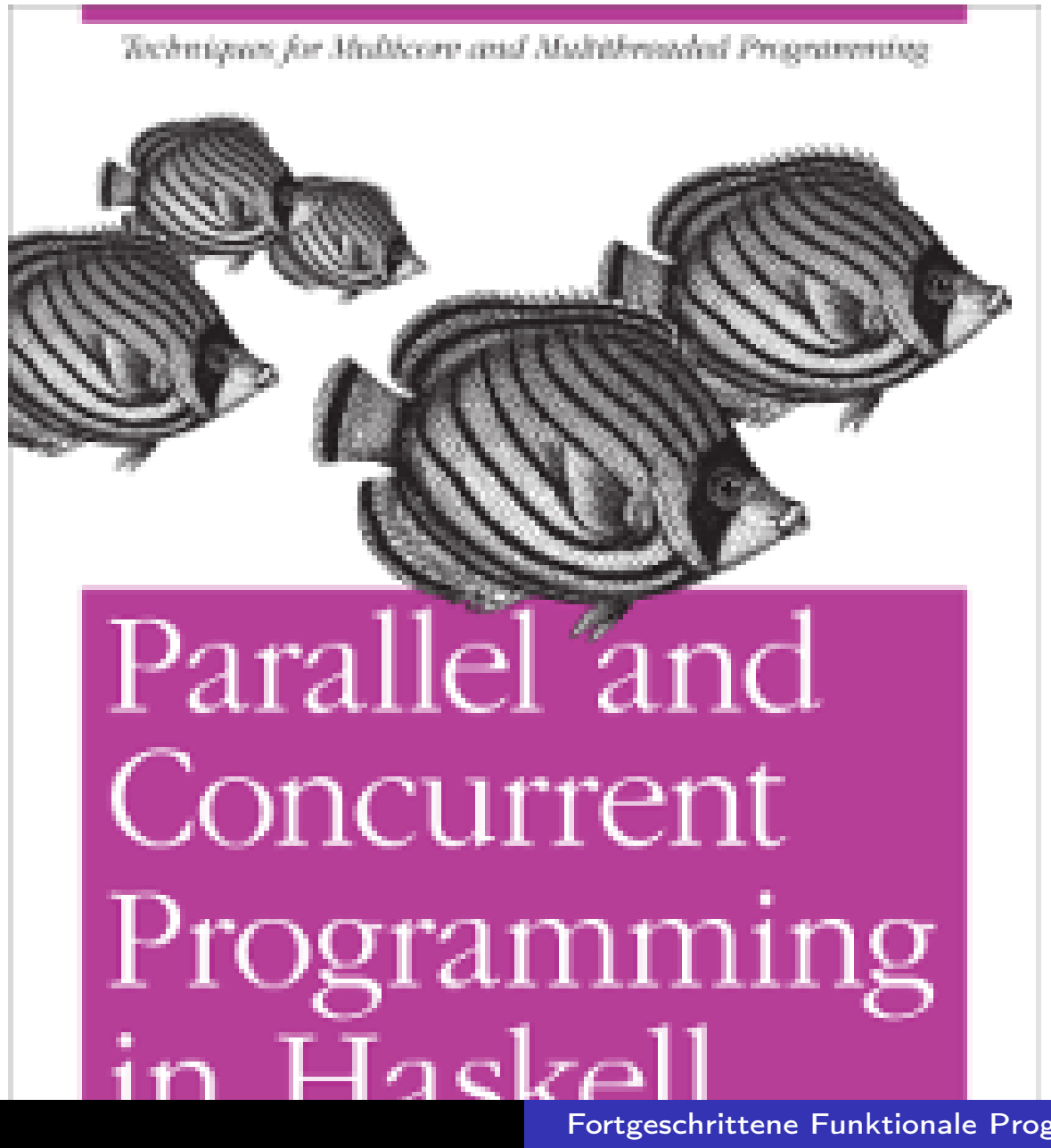
Leseempfehlung:



Leseempfehlung:



. . . srsly!


Techniques for Multicore and Multithreaded Programming
Parallel and
Concurrent
Programming
in Haskell

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 effektvoll ist, reicht `IO` vollkommen aus.

Unterschied zwischen `IVars` und `MVars`: erstere sind ***i***mmutable, letztere sind ***m***utable.

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.
- **Behältnis 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.
- **Behältnis 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**

Mehr Leckerlis:

Was passiert, wenn wir folgenden Code ausführen?

```
main :: IO ()
main = do m <- newEmptyMVar
          takeMVar m
```

Mehr Leckerlis:

Was passiert, wenn wir folgenden Code ausführen?

```
main :: IO ()
main = do m <- newEmptyMVar
          takeMVar m
```

Wir bekommen eine Fehlermeldung, dass das Programm hängt, statt einfach nur ein hängendes Programm.

```
$ ./mvar3
mvar3: thread blocked indefinitely in an MVar operation
```

Deadlock detection:

Threads und `MVars` sind Objekte auf dem Heap. Das `RTS` (i.e. der Garbage collector) durchläuft den Heap um alle lebendigen Objekte zu finden, angefangen bei den laufenden Threads und ihren Stacks.

Alles was so nicht erreichbar sind (z.B. ein Thread der auf eine `MVar` wartet, die nirgendwo sonst referenziert wird), blockiert und bekommt eine Exception geschmissen.

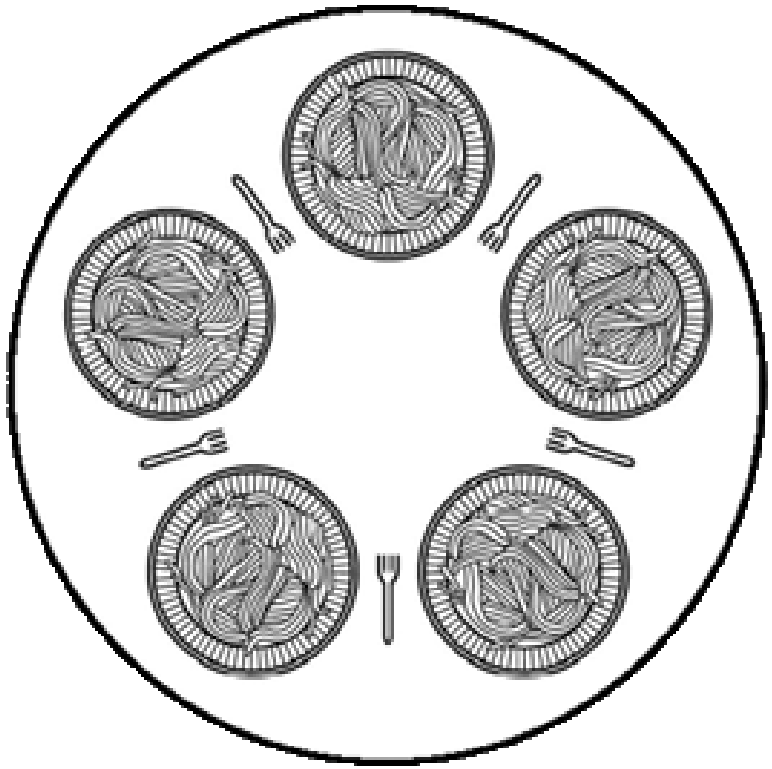


Abbildung: dining philosophers

Deadlock detection:

Dieses Vorgang funktioniert allerdings nicht immer wie man zunächst denkt. Beispiel: Was passiert mit diesem Code?

```
main :: IO ()
main = do
  lock <- newEmptyMVar
  complete <- newEmptyMVar
  forkIO $ takeMVar lock `finally` putMVar complete ()
  takeMVar complete
```

Deadlock detection:

Dieses Vorgang funktioniert allerdings nicht immer wie man zunächst denkt. Beispiel: Was passiert mit diesem Code?

```
main :: IO ()
main = do
  lock <- newEmptyMVar
  complete <- newEmptyMVar
  forkIO $ takeMVar lock `finally` putMVar complete ()
  takeMVar complete
```

Da nicht nur der geforkte Thread sondern auch der ursprüngliche gedeadlocked sind, wird hier die Fehlermeldung geprintet, statt die rettende Exception an das Kind zu senden.

Software Transactional Memory (STM)

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

Beliebte Fehler:

- **Races** (vergessene Locks)

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

Beliebte Fehler:

- **Races** (vergessene Locks)
- **Deadlocks** (Locks in falscher Reihenfolge genommen)

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

Beliebte Fehler:

- **Races** (vergessene Locks)
- **Deadlocks** (Locks in falscher Reihenfolge genommen)
- **Lost wakeups** (Conditional-Variable nicht bescheid gesagt)

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

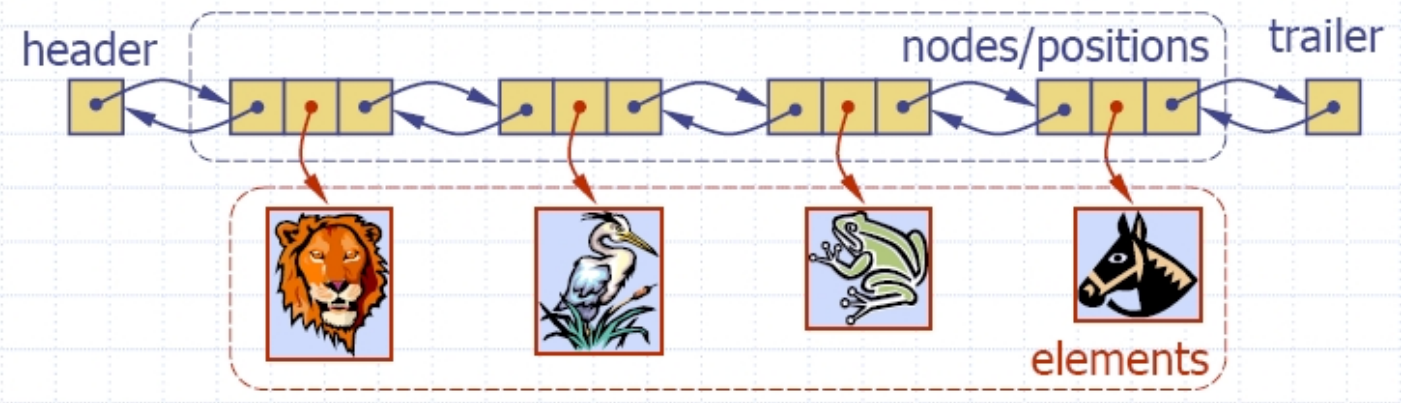
Beliebte Fehler:

- **Races** (vergessene Locks)
- **Deadlocks** (Locks in falscher Reihenfolge genommen)
- **Lost wakeups** (Conditional-Variable nicht bescheid gesagt)
- **Error Recovery** (Exceptionhandler müssen Locks freigeben und teilweise Ursprungszustand restaurieren)

Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

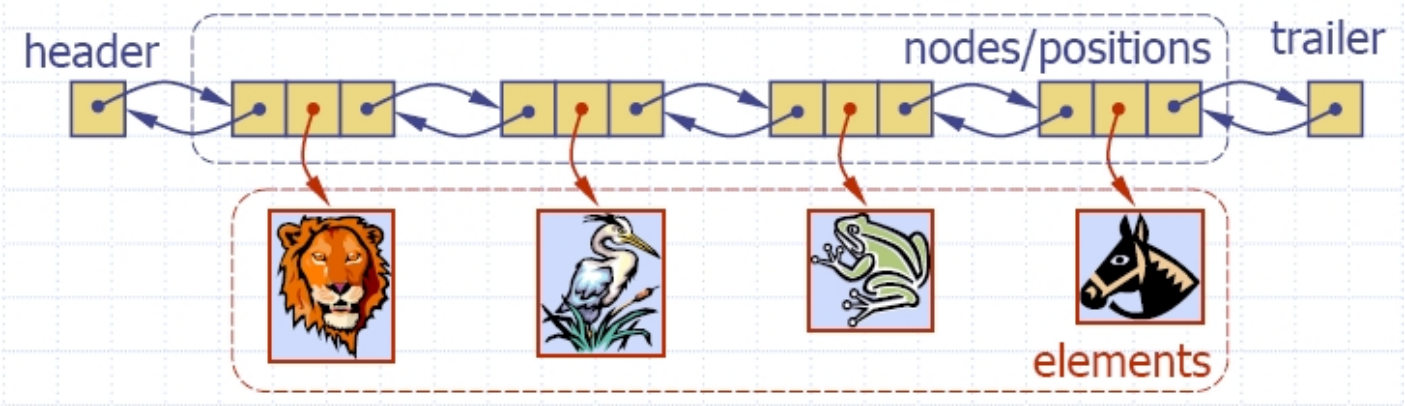
Bildquelle: http://www.math.bas.bg/~nkirov/2015/NETB201/slides/ch04/ch04.html



Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

Bildquelle: http://www.math.bas.bg/~nkirov/2015/NETB201/slides/ch04/ch04.html

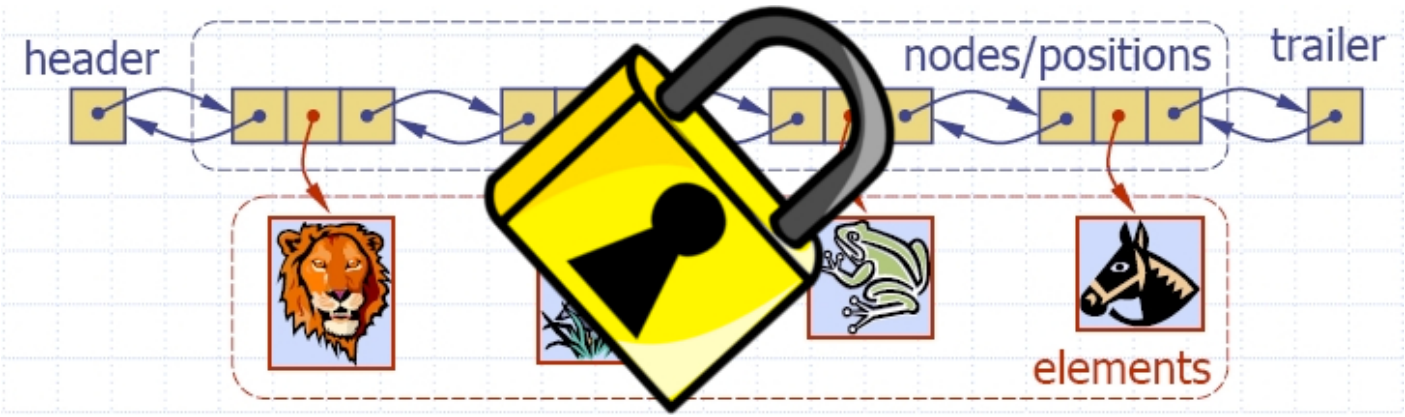


Problem: offensichtlich, race conditions etc.

Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

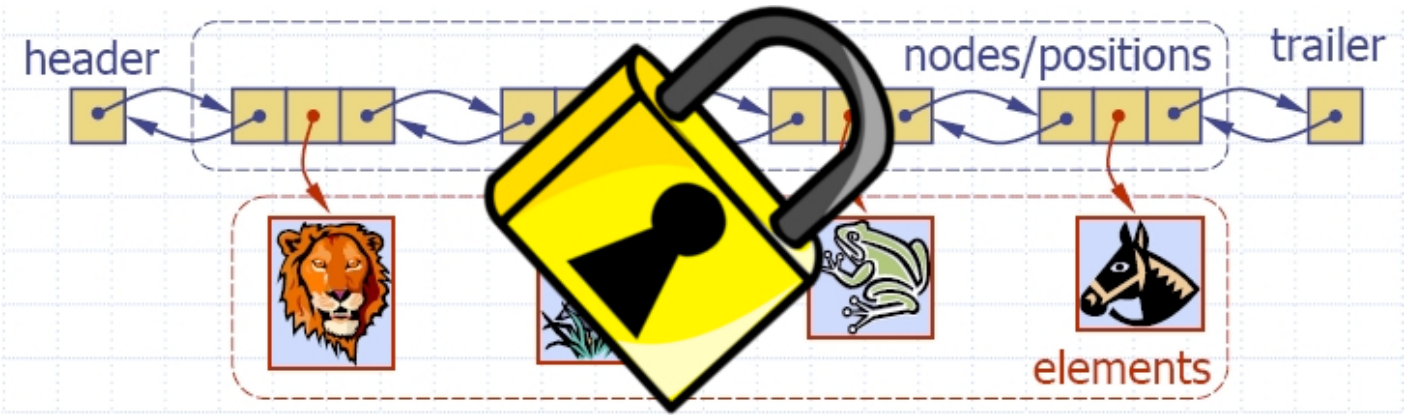
Bildquelle: http://www.math.bas.bg/∼nkirov/2015/NETB201/slides/ch04/ch04.html



Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

Bildquelle: http://www.math.bas.bg/∼nkirov/2015/NETB201/slides/ch04/ch04.html

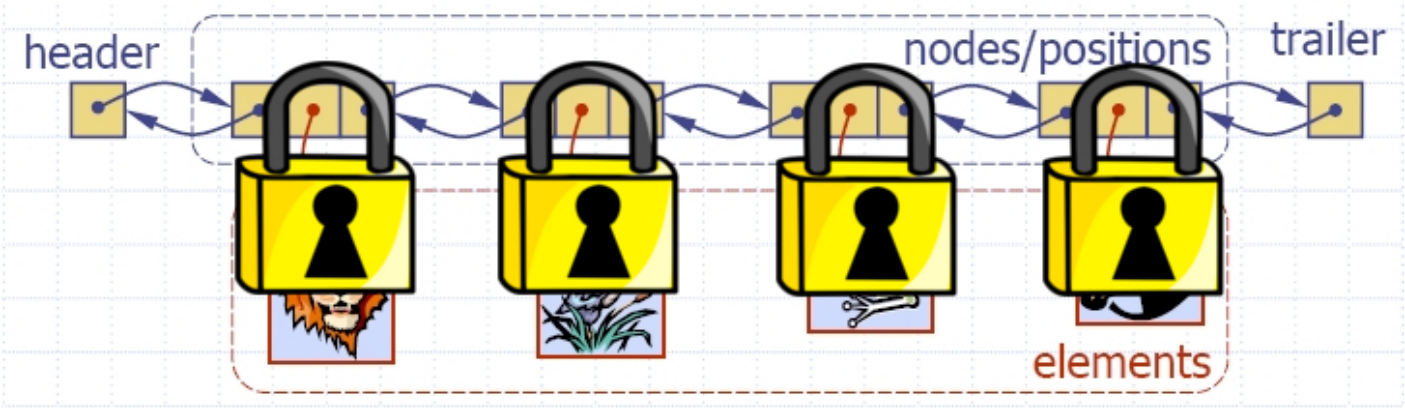


Problem: Nicht gerade nebenläufig. . .

Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

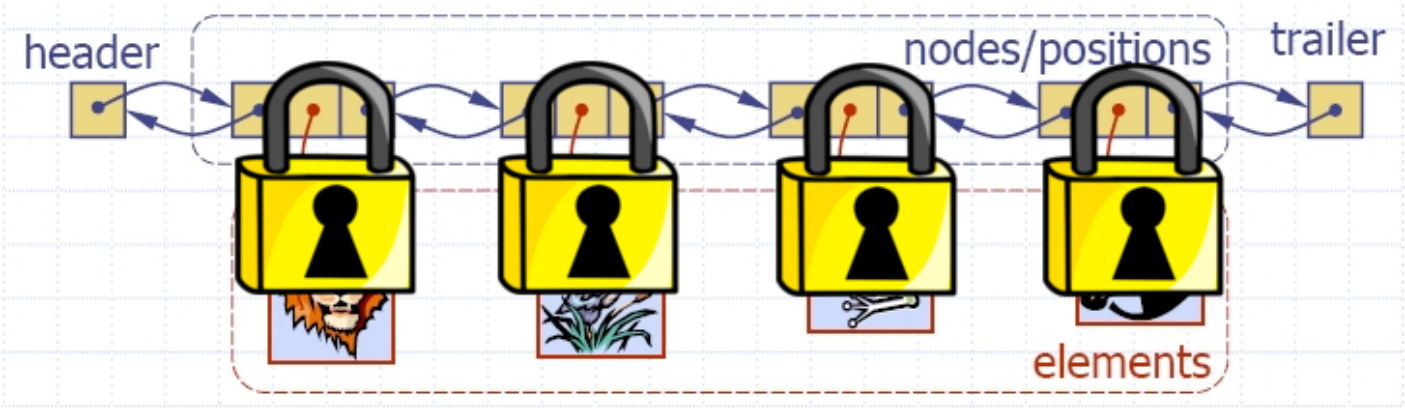
Bildquelle: http://www.math.bas.bg/∼nkirov/2015/NETB201/slides/ch04/ch04.html



Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

Bildquelle: http://www.math.bas.bg/∼nkirov/2015/NETB201/slides/ch04/ch04.html



Problem: Fehleranfälligkeit bei kleinen Listen

Aufgabe vs. Schwierigkeit:

Problemstellung	Schwierigkeit
sequentiell	Gymnasium oder Bachelor

Aufgabe vs. Schwierigkeit:

Problemstellung	Schwierigkeit
sequentiell	Gymnasium oder Bachelor
Locks et al.	(gerade nicht mehr) publizierbar auf internationalen Konferenzen

Aufgabe vs. Schwierigkeit:

Problemstellung	Schwierigkeit
sequentiell	Gymnasium oder Bachelor
Locks et al.	(gerade nicht mehr) publizierbar auf internationalen Konferenzen
atomic blocks (`STM`)	Bachelor

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren (d.h. nicht unterbrochenen) Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade auch (`do`-Notation etc.).

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren (d.h. nicht unterbrochenen) Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade auch (`do`-Notation etc.).

- Das bedeutet, dass Deadlocks unmöglich werden *weil es keine Locks mehr gibt*!

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren (d.h. nicht unterbrochenen) Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade auch (`do`-Notation etc.).

- Das bedeutet, dass Deadlocks unmöglich werden *weil es keine Locks mehr gibt*!
- Automatisierte error recovery. `STM` stellt den Ausgangszustand von selbst wieder her.

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren (d.h. nicht unterbrochenen) Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade auch (`do`-Notation etc.).

- Das bedeutet, dass Deadlocks unmöglich werden *weil es keine Locks mehr gibt*!
- Automatisierte error recovery. `STM` stellt den Ausgangszustand von selbst wieder her.
- `TVars` (Transaction Variables). Wie `IVars` und `MVars`, nur in der `STM`-Monade.

STM auf einen Blick:

```
data STM a                                  -- abstract
instance Monad STM                          -- among other things

atomically :: STM a -> IO a

data TVar a                                 -- abstract
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()

retry      :: STM a
orElse     :: STM a -> STM a -> STM a

throwSTM   :: Exception e => e -> STM a
catchSTM   :: Exception e => STM a -> (e -> STM a) -> STM a
```

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- . . . und zwar in einem Rutsch, ohne Unterbrechung!

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- . . . und zwar in einem Rutsch, ohne Unterbrechung!
- Stellt bei Fehlschlag Ursprungszustand wieder her.

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- . . . und zwar in einem Rutsch, ohne Unterbrechung!
- Stellt bei Fehlschlag Ursprungszustand wieder her.
- Deshalb: Kein `IO` in Transaktionen!

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- . . . und zwar in einem Rutsch, ohne Unterbrechung!
- Stellt bei Fehlschlag Ursprungszustand wieder her.
- Deshalb: Kein `IO` in Transaktionen!
- Ebenfalls: Keine genesteten `atomicallys` (Typen!)

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

- *Kein* Sprachkonstrukt!

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

- *Kein* Sprachkonstrukt!
- Rollt zurück und versucht die gleiche Transaktion erneut durchzuführen

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

- *Kein* Sprachkonstrukt!
- Rollt zurück und versucht die gleiche Transaktion erneut durchzuführen
- . . . allerdings erst zu einem angebrachten Zeitpunkt.
  CPU wird nicht unnützerweise zur Heizung.

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

- *Kein* Sprachkonstrukt!
- Rollt zurück und versucht die gleiche Transaktion erneut durchzuführen
- . . . allerdings erst zu einem angebrachten Zeitpunkt.
  CPU wird nicht unnützerweise zur Heizung.

Ein kurzer Blick auf `orElse`:

```
orElse :: STM a -> STM a -> STM a
```

Ein kurzer Blick auf `orElse`:

```
orElse :: STM a -> STM a -> STM a
```

- *Kein* Sprachkonstrukt!

Ein kurzer Blick auf `orElse`:

```
orElse :: STM a -> STM a -> STM a
```

- *Kein* Sprachkonstrukt!
- `orElse a b` führt `b` aus, wenn `a` `retry` aufruft.

Ein kurzer Blick auf `orElse`:

```
orElse :: STM a -> STM a -> STM a
```

- *Kein* Sprachkonstrukt!
- `orElse a b` führt `b` aus, wenn `a` `retry` aufruft.
- Komposition von `STM`-Berechnungen:
  `>>=` ist `AND`; `orElse` ist `OR`

Ein kurzer Blick auf `orElse`:

```
orElse :: STM a -> STM a -> STM a
```

- *Kein* Sprachkonstrukt!
- `orElse a b` führt `b` aus, wenn `a` `retry` aufruft.
- Komposition von `STM`-Berechnungen:
  `>>=` ist `AND`; `orElse` ist `OR`

Beispiel:

```
takeEitherTMVar :: TMVar a -> TMVar b -> STM (Either a b)
takeEitherTMVar ma mb =
  fmap Left (takeTMVar ma)
    `orElse`
  fmap Right (takeTMVar mb)
```

Stellen wir uns vor, wir wollen eine simplifizierte Bankensoftware schreiben, die in der Lage sein soll, Konten und Überweisungen zu repräsentieren.

Stellen wir uns vor, wir wollen eine simplifizierte Bankensoftware schreiben, die in der Lage sein soll, Konten und Überweisungen zu repräsentieren.

Eine naive Implementation wäre die folgende:

```
type Account = IORef Integer

transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    fromVal <- readIORef from
    toVal   <- readIORef to
    writeIORef from (fromVal - amount)
    writeIORef to (toVal + amount)
```

Diese Implementation hätte jedoch in einem Nebenläufigen Setting einige Probleme. Man beachte folgende Zeile:

```
fromVal <- readIORef from
```

Diese Implementation hätte jedoch in einem Nebenläufigen Setting einige Probleme. Man beachte folgende Zeile:

```
fromVal <- readIORef from
```

Finden nun mehrere Aktionen gleichzeitig statt, so könnte es sein, dass mehrere Threads denselben Wert als `fromVal` lesen, bevor die jeweils andere Transaktion durchgeführt wurde.
Dies hätte zur Folge, dass später ein inkorrekter Kontostand berechnet und gesetzt würde.

Führen wir also Locks ein (durch Benutzung von `MVars`), um diese race condition zu verhindern.

Führen wir also Locks ein (durch Benutzung von `MVars`), um diese race condition zu verhindern.

```haskell
type Account = MVar Integer

credit :: Integer -> Account -> IO ()
credit amount account = do
    current <- takeMVar account
    putMVar account (current + amount)

debit :: Integer -> Account -> IO ()
debit amount account = do
    current <- takeMVar account
    putMVar account (current - amount)
```

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Diese verhindert, dass durch fehlerhaftes Zusammenspiel von Threads Geld geschaffen oder vernichtet wird.

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Diese verhindert, dass durch fehlerhaftes Zusammenspiel von Threads Geld geschaffen oder vernichtet wird.

Es existiert allerdings immer noch eine race condition: Der Thread, der die Überweisung ausführt, könnte direkt nach dem `debit`-Schritt von der CPU verdrängt werden und die Bankensoftware dadurch in einem inkonsistenten Zustand zurück lassen.

Wie sähe diese Software mit `STM` aus?

```
type Account = TVar Integer

credit :: Integer -> Account -> STM ()
credit amount account = do
    current <- readTVar account
    writeTVar account (current + amount)

debit :: Integer -> Account -> STM ()
debit amount account = do
    current <- readTVar account
    writeTVar account (current - amount)

transfer :: Integer -> Account -> Account -> STM ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Vergleich zur Variante mit MVars:

```
type Account = MVar Integer

credit :: Integer -> Account -> IO ()
credit amount account = do
    current <- takeMVar account
    putMVar account (current + amount)

debit :: Integer -> Account -> IO ()
debit amount account = do
    current <- takeMVar account
    putMVar account (current - amount)

transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

```
transfer :: Integer -> Account -> Account -> IO ()
```

Diese Funktion führt die Überweisung vollkommen in der echten Welt durch, mit allen möglichen Fehlern, die dabei auftreten können.

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

```
transfer :: Integer -> Account -> Account -> IO ()
```

Diese Funktion führt die Überweisung vollkommen in der echten Welt durch, mit allen möglichen Fehlern, die dabei auftreten können.

```
transfer :: Integer -> Account -> Account -> STM ()
```

Diese Funktion stellt uns nur eine Berechnung in der `STM`-Monade bereit, die wir später entweder direkt oder auch als Baustein einer größeren Transaktion ausführen können.

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

```
transfer :: Integer -> Account -> Account -> IO ()
```

Diese Funktion führt die Überweisung vollkommen in der echten Welt durch, mit allen möglichen Fehlern, die dabei auftreten können.

```
transfer :: Integer -> Account -> Account -> STM ()
```

Diese Funktion stellt uns nur eine Berechnung in der `STM`-Monade bereit, die wir später entweder direkt oder auch als Baustein einer größeren Transaktion ausführen können.
Der Vorteil ist, dass wir nur einen Weg haben, diese Berechnung haben, das zu tun:

```
atomically :: STM a -> IO a
```

Parallelism through Concurrency