Concurrent Haskell und STM

Sébastien Braun

Proseminar Fortgeschrittene Programmierung
4. Februar 2008

Teil I

Concurrent Haskell

Inhalt Teil 1

Nebenläufigkeit

Definition (Nebenläufige Ausführung)

ist die gleichzeitige oder quasi-gleichzeitige Ausführung von mehreren Programmteilen oder Programmen.

Nebenläufigkeit

Definition (Nebenläufige Ausführung)

ist die gleichzeitige oder quasi-gleichzeitige Ausführung von mehreren Programmteilen oder Programmen.

Warum?

- Ausnutzung von Rechenkapazitäten (Multicore-Hardware, Hyperthreading)
- Modellierung von gleichzeitig ablaufenden Vorgängen. Für manche Probleme ist eine nebenläufige Formulierung natürlicher als eine sequentielle ($\rightarrow$ Netzwerkserver)

Überblick

Modul

```
import Control.Concurrent
```

Überblick

Modul

```
import Control.Concurrent
```

Starten eines Threads

```
forkIO :: IO () → IO ThreadId
```

Überblick

Modul

```
import Control.Concurrent
```

Starten eines Threads

```
forkIO :: IO () → IO ThreadId
```

Kommunikation zwischen Threads

- `MVars`

Überblick

Modul

```
import Control.Concurrent
```

Starten eines Threads

```
forkIO :: IO () → IO ThreadId
```

Kommunikation zwischen Threads

- `MVars`
- `Chans`

Überblick

Modul

```
import Control.Concurrent
```

Starten eines Threads

```
forkIO :: IO () → IO ThreadId
```

Kommunikation zwischen Threads

- `MVars`
- `Chans`
- Software Transactional Memory (Teil 2)

Threads starten

forkIO

```
forkIO :: IO () → IO ThreadId
```

Threads starten

forkIO

```
forkIO :: IO () → IO ThreadId
```

Beispiel

```
putChars :: Char →  IO ()
putChars c = do putChar c
                putChars c

main :: IO ()
main = do forkIO (putChars 'a')
          putChars 'b'
```

MVars

Definition (MVar)

`data MVar` α `= <abstrakt>`

Veränderliche Speicherzelle, die leer sein kann.

MVars

Definition (MVar)

`data MVar` α `= <abstrakt>`

Veränderliche Speicherzelle, die leer sein kann.

Primitive auf MVars

- *newMVar* `::` $\alpha \rightarrow$ `IO (MVar` α `)`: MVar mit Anfangswert.
- *newEmptyMVar* `:: IO (MVar` α `)`: leere MVar.

MVars

Definition (MVar)

`data MVar` α `= <abstrakt>`

Veränderliche Speicherzelle, die leer sein kann.

Primitive auf MVars

- *newMVar* `::` $\alpha \rightarrow$ `IO (MVar` α `)`: MVar mit Anfangswert.
- *newEmptyMVar* `:: IO (MVar` α `)`: leere MVar.
- *takeMVar* `:: MVar` $\alpha \rightarrow$ `IO` α : wartet, bis ein Wert in der MVar ist, und entnimmt ihn.

MVars

Definition (MVar)

`data MVar` α `= <abstrakt>`

Veränderliche Speicherzelle, die leer sein kann.

Primitive auf MVars

- *newMVar* `::` $\alpha \rightarrow$ `IO (MVar` α `)`: MVar mit Anfangswert.
- *newEmptyMVar* `:: IO (MVar` α `)`: leere MVar.
- *takeMVar* `:: MVar` $\alpha \rightarrow$ `IO` α : wartet, bis ein Wert in der MVar ist, und entnimmt ihn.
- *putMVar* `:: MVar` $\alpha \rightarrow \alpha \rightarrow$ `IO ()`: legt einen Wert in eine *leere* MVar.

MVars: Ein Beispiel

```
consumer :: MVar Int → IO ()

consumer mvar =
  do putStrLn $ "Empfange einen Wert"
     x ← takeMVar mvar
     putStrLn $ "Empfangen: " ++ (show x)
```

```
producer :: MVar Int → IO ()

producer mvar =
  do putStrLn $ "Sende einen Wert"
     putMVar mvar 1
     putStrLn $ "Gesendet: 1"
```

main :: IO ()

```
main =
  do mvar ← newEmptyMVar
     forkIO (consumer mvar)
     producer mvar
```

Chan

Was ist ein Chan?

Ein `Chan` α ist ein gepufferter Kanal von Elementen des Typs α, der ein Lese-Ende und ein Schreib-Ende besitzt. Die Elemente werden in FIFO-Reihenfolge verarbeitet.

Chan

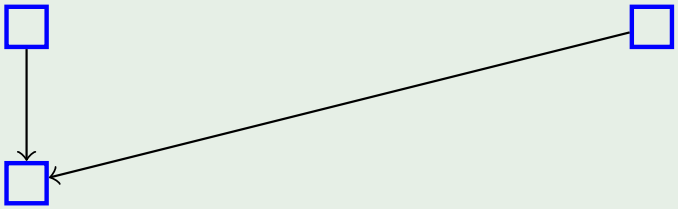
Definition (Chan)

```
type Chan α  =
 (MVar (Stream α ),
  MVar (Stream α ))
type Stream α  =
  MVar (Item α )
data Item α  =
  Item a (Stream α )
```

Ein Beispiel

Beispiel (Einfügeoperationen)

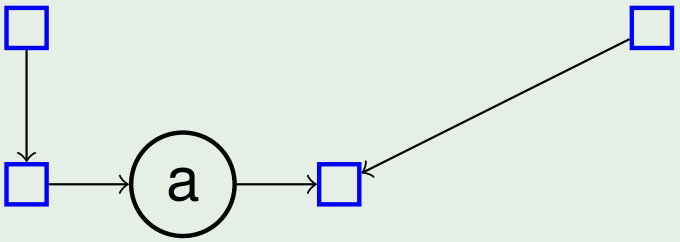
Leer:



Ein Beispiel

Beispiel (Einfügeoperationen)

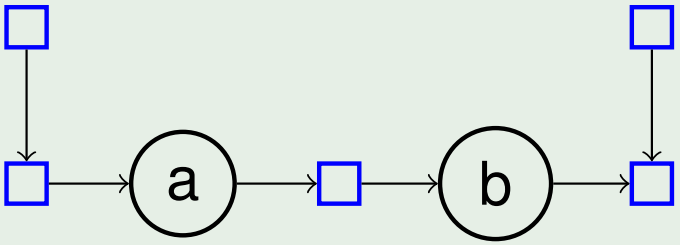
'a' einfügen:



Ein Beispiel

Beispiel (Einfügeoperationen)

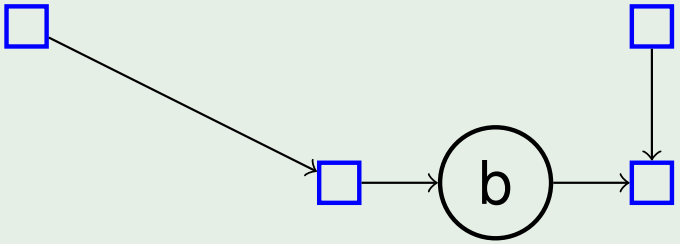
’b’ einfügen:



Ein Beispiel

Beispiel (Löschoperationen)

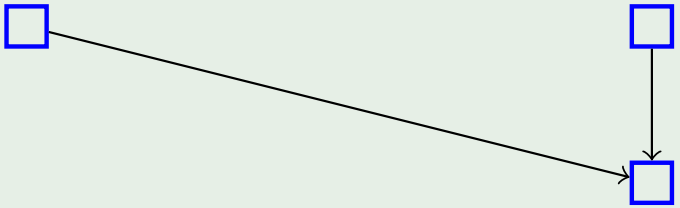
'a' entnehmen:



Ein Beispiel

Beispiel (Löschoperationen)

'b' entnehmen:



Funktionen auf `Chans`

Modul

```
import Control.Concurrent.Chan
```

Funktionen

`newChan :: IO (Chan α )` Erzeugt einen neuen Chan

`writeChan :: Chan α → α → IO ()` Hängt einen Wert an

`readChan :: Chan α → IO α` Holt den ersten Wert

Funktionen auf `Chans`

Modul

```
import Control.Concurrent.Chan
```

Funktionen

newChan :: IO (Chan α) Erzeugt einen neuen Chan

writeChan :: Chan $\alpha \rightarrow \alpha \rightarrow$ IO () Hängt einen Wert an

readChan :: Chan $\alpha \rightarrow$ IO α Holt den ersten Wert

Weitere nützliche Funktionen in `Control.Concurrent.Chan`

Zur Erinnerung: Semaphore

Definition (Semaphor)

Ein Semaphor ist eine ganzzahlige veränderliche Variable n mit zwei Operationen:

Zur Erinnerung: Semaphore

Definition (Semaphor)

Ein Semaphor ist eine ganzzahlige veränderliche Variable n mit zwei Operationen:

`P()` Warte, bis $n > 0$ und setze dann $n \leftarrow n - 1$

Zur Erinnerung: Semaphore

Definition (Semaphor)

Ein Semaphor ist eine ganzzahlige veränderliche Variable n mit zwei Operationen:

- `P()` Warte, bis $n > 0$ und setze dann $n \leftarrow n - 1$
- `V()` Setze $n \leftarrow n + 1$ und wecke wartende Prozesse auf.

Können wir mit MVars Semaphore simulieren?

Ja!

Können wir mit MVars Semaphore simulieren?

Ja!

Definition (P())

```
type QSem = MVar (Int, [MVar ()])
waitQSem qsem = do
  (available, blocked) ←  takeMVar qsem
```

Können wir mit MVars Semaphore simulieren?

Ja!

Definition (P())

```
type QSem = MVar (Int, [MVar ()])
waitQSem qsem = do
  (available, blocked) ← takeMVar qsem
  if available > 0 then
    putMVar qsem (available - 1, blocked)
```

Können wir mit MVars Semaphore simulieren?

Ja!

Definition (P())

```
type QSem = MVar (Int, [MVar ()])
waitQSem qsem = do
  (available, blocked) ←  takeMVar qsem
  if available > 0 then
    putMVar qsem (available - 1, blocked)
   else do
    blockOn ←  newEmptyMVar
    putMVar qsem (available, blocked ++ [blockOn])
    takeMVar blockOn
```

Können wir mit MVars Semaphore simulieren?

Ja!

Definition (V())

```
type QSem = MVar (Int, [MVar ()])
signalQSem qsem = do
  (available, blocked) ←  takeMVar qsem
```

Können wir mit MVars Semaphore simulieren?

Ja!

Definition (V())

```
type QSem = MVar (Int, [MVar ()])
signalQSem qsem = do
  (available, blocked) ← takeMVar qsem
  case blocked of
    [] → putMVar qsem (available + 1, [])
```

Können wir mit MVars Semaphore simulieren?

Ja!

Definition (V())

```
type QSem = MVar (Int, [MVar ()])
signalQSem qsem = do
  (available, blocked) ← takeMVar qsem
  case blocked of
    [] → putMVar qsem (available + 1, [])
    (b:bs) → do
      putMVar qsem (available, bs)
      putMVar b ()
```

Können wir mit MVars Semaphore simulieren?

In der Standardbibliothek

Die Haskell-Standardbibliothek enthält bereits eine Implementierung von Semaphoren.

Eine neuer Semaphor mit Anfangswert kann mittels
`newQSem :: Int → IO QSem`
angelegt werden.

Semaphore sind nicht optimal

Warum nicht?

- Deadlocks
- Prioritätsumkehr
- ...

Semaphore sind nicht optimal

Warum nicht?

- Deadlocks
- Prioritätsumkehr
- ...

Ursache

Ursache für viele Probleme: wechselseitiger Ausschluss. Geht es auch ohne?

Teil II

Software Transactional Memory

Inhalt Teil 1

Transaktionen

Transaktionen

ACID

Atomicity	Eine Transaktion läuft entweder ganz oder gar nicht ab.
Consistency	Eine Transaktion kann nicht die „Regeln” brechen.
Isolation	Keine andere Transaktion kann einen Zwischenstand einsehen.
Durability	Die Ergebnisse einer Transaktion sind dauerhaft.

Transaktionen

ACID

Atomicity Eine Transaktion läuft entweder ganz oder gar nicht ab.

Consistency Eine Transaktion kann nicht die „Regeln” brechen.

Isolation Keine andere Transaktion kann einen Zwischenstand einsehen.

Wofür Transaktionen?

Vorteile

- Bewährtes Konzept für gleichzeitigen Zugriff auf Daten.
- Abstraktionsebene: Wir geben an, dass Daten vor Kollisionen zu schützen sind.
  Wie das passiert, überlassen wir dem System.

Wofür Transaktionen?

Vorteile

- Bewährtes Konzept für gleichzeitigen Zugriff auf Daten.
- Abstraktionsebene: Wir geben an, dass Daten vor Kollisionen zu schützen sind.
  Wie das passiert, überlassen wir dem System.

Frage:

Kann man das Transaktionskonzept auch auf Variable im Speicher anwenden?

Transactional Memory

Was ist Transactional Memory?

> *[Transactional Memory is] a new multiprocessor architecture intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion.”*

(Herlihy, 1993)

Transactional Memory

Was ist Transactional Memory?

> *[Transactional Memory is] a new multiprocessor architecture intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion.”*

(Herlihy, 1993)

Software Transactional Memory

Software Transactional Memory ist eine Haskell-Implementierung dieser Idee.

TVars

Definition (TVar)

`data TVar` α `= ...abstrakt...`

Veränderliche Speicherzelle, nur in der STM-Monade zugänglich.

TVars

Definition (TVar)

`data TVar` α `= ...abstrakt...`
Veränderliche Speicherzelle, nur in der STM-Monade zugänglich.

Operationen auf TVars

- `newTVar :: ` $\alpha \rightarrow$ `STM (TVar` α `)` Neue TVar mit Anfangswert
- `readTVar :: TVar` $\alpha \rightarrow$ `STM` α Liest eine TVar
- `writeTVar :: TVar` $\alpha \rightarrow \alpha \rightarrow$ `STM ()` Schreibt einen Wert in eine TVar.

atomically

Von der STM- in die IO-Monade

```
f =                do x ←  readTVar tvar1
                      writeTVar tvar2 y
```

f hat den Typ `STM ()`. Wir brauchen aber `IO ()`!

atomically

Von der STM- in die IO-Monade

```
f = atomically (do x ←  readTVar tvar1
                   writeTVar tvar2 y)
```

`atomically :: STM` $\alpha \rightarrow$ `IO` α sorgt dafür, dass alle TVar-Zugriffe unteilbar („atomically”) ablaufen und „holt die Berechnung aus der STM-Monade heraus”.

Transaktionen

Frage:

Was passiert wenn ein Thread eine TVar liest, während sie von einem anderen Thread geändert wird?

Transaktionen können fehlschlagen

Frage:

Was passiert wenn ein Thread eine TVar liest, während sie von einem anderen Thread geändert wird?

Antwort:

Die Transaktion schlägt fehl.

- Haskell führt über `TVar`-Zugriffe Buch.
- Bei Konflikt:
  - Beteiligte Transaktionen werden abgebrochen
  - Und neu gestartet.

retry

Definition

retry :: STM α

Ein Aufruf von *retry* bewirkt, dass die laufende Transaktion als fehlgeschlagen behandelt wird.

retry

Definition

retry :: STM α
Ein Aufruf von *retry* bewirkt, dass die laufende Transaktion als fehlgeschlagen behandelt wird.

Verwendung

Transaktionen können auch durch Umstände fehlschlagen, die das Haskell-System nicht selbstständig erkennen kann.

Z.B.: Es liegt noch kein Wert vor, mit dem gearbeitet werden kann.

orElse

Definition

$\texttt{orElse} :: \texttt{STM}\ \alpha \rightarrow \texttt{STM}\ \alpha \rightarrow \texttt{STM}\ \alpha$

`a` ‘*orElse*‘ `b`: versucht `a` und führt `b` aus, wenn `a` fehlschlägt.

orElse

Definition

`orElse :: STM` $\alpha \rightarrow$ `STM` $\alpha \rightarrow$ `STM` α

`a` ‘`orElse`‘ `b`: versucht `a` und führt `b` aus, wenn `a` fehlschlägt.

Verwendung

Mit ‘`orElse`‘ werden Transaktionen kombiniert. Der Vorteil von `orElse` ist, dass die einzelnen Transaktionen keine besonderen Vorkehrungen treffen müssen, um kombinierbar zu sein.

MVars in STM

Definition (*takeTMVar* :: TMVar α → STM α)

```
type TMVar α  = TVar (Maybe α )
takeTMVar tmvar =
  do v ←  readTVar tmvar
     case v of
       Nothing →  retry
       Just v' →  do writeTVar tmvar Nothing
                     return v'
```

MVars in STM

Definition (*putTMVar* :: TMVar $\alpha \rightarrow \alpha \rightarrow$ STM ())

```
type TMVar α  = TVar (Maybe α )
putTMVar tmvar a =
  do v ←  readTVar tmvar
     case v of
       Nothing →  writeTVar tmvar (Just a)
       Just _ →  retry
```

orElse und TMVars

Definition (*tryPutTMVar* :: TMVar $\alpha \rightarrow \alpha \rightarrow$ STM Bool)

```
type TMVar α  = TVar (Maybe α )
tryPutTMVar tmvar a =
  (do putTMVar tmvar a
      return True)
  `orElse`
  return False
```

orElse und TMVars

Definition (*tryPutTMVar* :: TMVar $\alpha \rightarrow \alpha \rightarrow$ STM Bool)

```
type TMVar α  = TVar (Maybe α )
tryPutTMVar tmvar a =
  (do putTMVar tmvar a
      return True)
  `orElse`
  return False
```

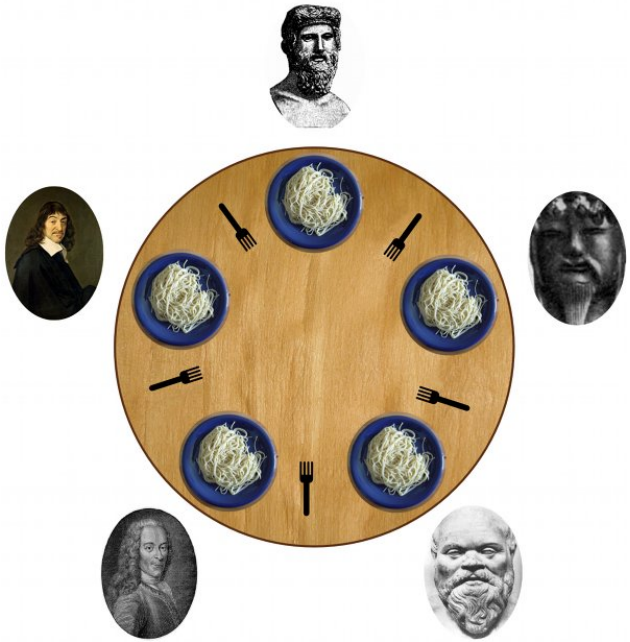
Der Mehrwert von orElse

Diese Definition ist ohne *orElse* nicht ohne Weiteres möglich.

Zum Auffrischen: Die speisenden Philosophen

Das Problem

5 Philosophen sitzen im Kreis um einen runden Tisch.
Die Philosophen diskutieren. Wenn ein Philosoph hungrig wird, nimmt er „seine” Gabeln vom Tisch und beginnt zu essen. Hat er genug gegessen, legt er die Gabeln zurück.
Wie kann man sicherstellen, dass kein Philosoph verhungert?



Essen

Code

```
essen :: Int → Gabeln → IO ()
essen n tisch =
  do putStrLn $ "Philosoph " ++ (show n) ++ " hungrig"
     atomically (holeGabeln n tisch)
     putStrLn $ "Philosoph " ++ (show n) ++ " isst"
     randomIO >>= (threadDelay . (`mod`20000000))
     atomically (legeGabeln n tisch)
```

Gabeln nehmen

Code

```
holeGabeln :: Int → Gabeln → STM ()
holeGabeln n tisch =
  do holeGabel n tisch
     holeGabel ((n+1) `mod` (length tisch)) tisch

holeGabel :: Int → Gabeln → STM ()
holeGabel n tisch =
  do liegt ← readTVar (tisch !! n)
     if liegt then writeTVar (tisch !! n) False
     else retry
```

Gabeln zurücklegen

Code

```
legeGabeln :: Int → Gabeln → STM ()
legeGabeln n tisch =
  do writeTVar (tisch !! n) True
     writeTVar (tisch !!
                  ((n + 1) `mod` (length tisch))
                ) True
```

Teil III

Zusammenfassung

Zusammenfassung

Concurrent Haskell

- ermöglicht das Verfassen von nebenläufigem Code in Haskell
- Mächtiger Synchronisationsmechanismus: `MVars`
- Chans bieten eine weitere Kommunikationsmöglichkeit
- Kommunikation mittels MVars und Chans ist automatisch synchronisiert.

Zusammenfassung

Software Transactional Memory

- erweitert Concurrent Haskell um Kommunikation und Synchronisation ohne wechselseitigen Ausschluss
- Keine Deadlocks und Prioritätsumkehr
- Transaktionen sind kombinierbar — Fehlgeschlagene Transaktionen können noch „gerettet” werden

Danke für die Aufmerksamkeit!

Danke für die Aufmerksamkeit!

Literatur

- Peyton Jones *et al.*, Concurrent Haskell, `http://citeseer.ist.psu.edu/jones96concurrent.html`
- Harris *et al.*, Composable Memory Transactions, `http://www.haskell.org/~simonmar/papers/stm.pdf`