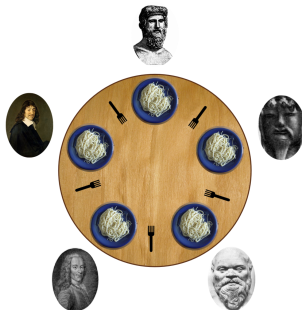
Concurrent Haskell

Bertram, Alexander & Wenzel, Sebastian

Fachhochschule Wedel - University of Applied Sciences

18. Dezember 2009



Gliederung

1. Nebenläufigkeit
2. Concurrent Haskell
3. Software Transactional Memory

Gliederung

1. Nebenläufigkeit
   - Grundlagen
   - Typische Probleme
2. Concurrent Haskell
   - Grundlagen
   - Threads
   - Synchronisation
   - Kommunikation
3. Software Transactional Memory

Kurzdefinition: Nebenläufigkeit

Mehrere Ereignisse sind nebenläufig, wenn sie sich nicht gegenseitig beeinflussen. Kein Ereignis darf Ursache eines anderen Ereignisses sein.

Grundlagen

Nebenläufigkeit in der Informatik

- Aufteilung der Programmfunktionalität auf separate Prozesse oder Threads
- Synchronisation
  - Semaphoren
  - Locks
  - Monitore
- Kommunikation
  - Message Passing
  - Mailboxen
- Probleme
  - Deadlocks
  - Verhungern

Typische Probleme

Erzeuger-Verbraucher

- Ein Erzeuger
- Ein Verbraucher
- Begrenzter Puffer
- Problem, wenn mehr erzeugt / verbraucht als verbraucht / erzeugt wird

Dinierende Philosophen

- n Philosophen, die gelegentlich essen
- 2 Gabeln, links und rechts, zum Essen nötig
- Problem: nur n Gabeln vorhanden

Typische Probleme

Leser-Schreiber

- n Schreib-Prozesse
- m Lese-Prozesse
- Ein Datenbestand
- Problem, wenn gleichzeitig geschrieben / gelesen wird

Gliederung

1. Nebenläufigkeit
   - Grundlagen
   - Typische Probleme

2. Concurrent Haskell
   - Grundlagen
   - Threads
   - Synchronisation
   - Kommunikation

3. Software Transactional Memory

Grundlagen

- Erweiterung zu Haskell 98
- Einführung des Nebenläufigkeit-Konzeptes in Haskell
- Bibliothek `Control.Concurrent` muss importiert werden
- Von GHC und Hugs unterstützt
- In GHC und Hugs teilweise unterschiedlich implementiert
- GHC bietet umfangreichere Unterstützung als Hugs

Concurrent.hs

```
...
#ifdef __GLASGOW_HASKELL__
  ...
#endif
...
```

Grundideen

- Thread
- Thread-Erzeugung
- Möglichkeiten zur Prozesssynchronisation und -kommunikation

Threads

GHC vs. Hugs

- GHC: Unterteilung in Haskell- und Betriebssystem-Threads
- Hugs: Nur Haskell-Threads

- Threads sind `IO`-Aktionen
- Ausführung sofort nach Erzeugung
- Thread zu Ende, wenn `IO`-Aktion zu Ende
- Nicht-deterministisch
- Thread-Zustand nach außen nicht sichtbar
- `ThreadId`
  - Abstrakter Datentyp
  - GHC: Handle des erzeugten Thread
  - Hugs: Synonym für ()

Haskell-Threads

- „Leichtgewichtig“
  - Overhead für die Erzeugung sehr klein
  - Overhead für den Kontextwechsel sehr klein
- Scheduling intern in der Haskell-Laufzeitumgebung
  - Keine Betriebssystem-Bibliotheken nötig

Erzeugung

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

- Scheduling
  - GHC: Präemptiv
  - Hugs: Kooperativ

forkIO-Beispiel

```
forkIO (write (take 10 (repeat 'a'))) >>
  write (take 10 (repeat 'b'))
  where
    write [] = putChar '.'
    write (c:cs) = putChar c >> write cs
```

Haskell-Threads

GHC

```
*Main> main
bbbbbbbbbb.aaaaaaaaaa.
*Main> main
bbbabababababababa.aa.
*Main> main
bbbabababababababa.aa.
```

Hugs

```
Main> main
bbbbbbbbbb.aaaaaaaaaa.
Main> main
bbbbbbbbbb.aaaaaaaaaa.
Main> main
bbbbbbbbbb.aaaaaaaaaa.
```

Betriebssystem-Threads

Erzeugung

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

- Erstellung über Betriebssystem-Funktionen
- Verwaltung vom Betriebssystem
- Erstellen und Kontextwechsel teuer
- Verwendung von Fremdbibliotheken möglich, in denen Threads mit Thread-lokalen Speicher benötigt werden
- Option `-threaded` beim Linken nötig

Option `-threaded`

```
$ ghc -c Main.hs
$ ghc -threaded -o Main Main.io
```

Blockierverhalten von Haskell-Threads

Problem

- Haskell-Laufzeitumgebung verwendet nur einen Betriebssystem-Thread
- Nur ein Haskell-Thread zur Zeit
- Blockierende Operationen blockieren alle anderen Haskell-Threads
  - Ausnahme: I/O-Operationen in GHC

Lösung

- Nur im GHC
- Option `-threaded` beim Linken nötig
- Verwendung der Multithread-Version der Haskell-Laufzeitumgebung

Gebundene Threads

- Bindung eines Haskell-Threads an einen Betriebssystem-Thread
- Verwaltung des Haskell-Threads von der Haskell-Laufzeitumgebung
- Verwendund des Betriebssystem-Thread für die Kommunikation mit Fremdbibliotheken
- Main-Thread ist immer ein gebundener Thread
- Option `-threaded` beim Linken nötig

Erzeugung

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

Concurrent.hs

Zusätzliche Funktionen

```
myThreadId :: IO ThreadId
killThread :: ThreadId -> IO ()
yield :: IO ()
threadDelay :: Int -> IO ()
rtsSupportsBoundThreads :: Bool
isCurrentThreadBound :: IO Bool
runInBoundThread :: IO a -> IO a
runInUnboundThread :: IO a -> IO a
```

Die meisten der genannten Funktionen sind nur im GHC verfügbar.

Synchronisation

- Möglich durch sogenannte „Mutable Variable“
- Entweder leer oder voll

Definition

```
data MVar a
```

Operationen

- `takeMVar :: MVar a -> IO a`
- `putMVar :: MVar a -> a -> IO ()`

„Sterbende“ Philosophen

```
philosoph :: Int -> MVar () -> MVar () -> IO ()

philosoph n left right = do
  takeMVar left
  takeMVar right
  -- eat
  putMVar left ()
  putMVar right ()
  philosoph n left right
```

Dinierende Philosophen

```haskell
philosoph :: Int -> MVar () -> MVar () -> IO

philosoph n left right = do
  takeMVar left
  empty <- isEmptyMVar right
  if empty then
    putMVar left ()
    philosoph n left right
  else do
    takeMVar right
    -- eat
    putMVar left ()
    putMVar right ()
    philosoph n left right
```

Kommunikation

- Auch über MVars
- Einelementiger Puffer

Operationen

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

- Im GHC auch über Exceptions möglich

Kommunikation - Beispiel

Erzeuger

```
producer :: MVar Int -> IO ()
producer mVar = do
  putStrLn $ "Sende einen Wert"
  putMVar mVar 42
  putStrLn $ "Gesendet: 42"
```

Verbraucher

```
consumer :: MVar Int -> IO ()
consumer mMar = do
  putStrLn $ "Warte auf einen Wert"
  x <- takeMVar mVar
  putStrLn $ "Empfangen: " ++ (show x)
```

Kommunikation - Beispiel

Aufruf

```
startProducerConsumer = do
  mVar <- newEmptyMVar
  forkIO (producer mVar)
  consumer mVar
```

Ausgabe

```
*Main> startProducerConsumer
Verbraucher: Warte auf einen Wert
Erzeuger: Sende 42
Erzeuger: 42 gesendet
Verbraucher: 42 empfangen
```

MVar.hs

Zusätzliche Funktionen

```
readMVar :: MVar a -> IO a
swapMVar :: MVar a -> a -> IO a
tryTakeMVar :: MVar a -> IO (Maybe a)
tryPutMVar :: MVar a -> a -> IO Bool
isEmptyMVar :: MVar a -> IO Bool
withMVar :: MVar a -> (a -> IO b) -> IO b
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
```

Kommunikation über Channels

Definition

```
data Chan a =
  Chan (MVar (Stream a)) -- read end
       (MVar (Stream a)) -- write end

type Stream a = MVar (ChItem a)

data ChItem a = ChItem a (Stream a)
```

- Abstrakter Datentyp
- Repräsentiert unbegrenzten FIFO Puffer
- Ermöglicht synchrones Schreiben und Lesen

Chan.hs

Operationen

```
newChan :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan :: Chan a -> IO a
dupChan :: Chan a -> IO (Chan a)
unGetChan :: Chan a -> a -> IO ()
isEmptyChan :: Chan a -> IO Bool
getChanContents :: Chan a -> IO [a]
writeList2Chan :: Chan a -> [a] -> IO ()
```

Gliederung

1. Nebenläufigkeit
   - Grundlagen
   - Typische Probleme
2. Concurrent Haskell
   - Grundlagen
   - Threads
   - Synchronisation
   - Kommunikation
3. Software Transactional Memory

Probleme mit nebenläufigen Programmen

Synchronisation und Kommunikation

- Zu wenige Locks
- Zu viele Locks
- Falsche Locks
- Reihenfolge der Locks

Lösung: Transaktion

- Atomarität
- Kontinuität
- Isolation

STM-Monade

- Kann mit `atomically` mehrere `STM`-Aktionen sequentiell ausführen

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

- Arbeitet mit `TVars`

```
data TVar a
```

```
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Beispiel

Bank-Konten

```
type Account = TVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = atomically (do {
  deposit to amount;
  withdraw from amount})

withdraw :: Account -> Int -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)

deposit :: Account -> Int -> STM ()
deposit acccount amount =
  withdraw account (-amount)
```

Ende

Vielen Dank für die Aufmerksamkeit!
Schöne Feiertage!
Frohes Fest!
Und einen Guten Rutsch!