Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 01.12.09:
Grundlagen der Nebenläufigkeit in Haskell

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

1

Fahrplan

- ▶ Teil I: Monaden und fortgeschrittene Typen
- ▶ Teil II: Fortgeschrittene Datenstrukturen
- ▶ Teil III: Nebenläufigkeit
 - ▶ Grundlagen
 - ▶ Abstraktionen und Ausnahmebehandlung
 - ▶ Software Transactional Memory
- ▶ Teil IV: The Future of Programming

2

Heute gibt's hier

Nebenläufigkeit

- ▶ Grundkonzepte
- ▶ Implementation in Haskell
- ▶ Basiskonzepte

3

Konzepte der Nebenläufigkeit

Thread (lightweight process)	vs.	Prozess
Programmiersprache/Betriebssystem
(z.B. Java, Haskell, Linux)		Betriebssystem
gemeinsamer Speicher		getrennter Speicher
Erzeugung billig		Erzeugung teuer
mehrere pro Programm		einer pro Programm
- ▶ Multitasking:
 - ▶ präemptiv: Kontextwechsel wird erzwungen
 - ▶ kooperativ: Kontextwechsel nur freiwillig

4

Zur Erinnerung: 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 synchronize

5

Threads in Haskell: Concurrent Haskell

- ▶ Sequentielles Haskell: Reduktion eines Ausdrucks
 - ▶ Compiler legt Reihenfolge fes (outermost leftmost — verzögerte Auswertung)
- ▶ Nebenläufiges Haskell: Reduktion eines Ausdrucks an mehreren Stellen
- ▶ ghc und hugs implementieren Haskell-Threads
- ▶ ghc: präemptiv, hugs: kooperativ
- ▶ Modul Control.Concurrent enthält Basisfunktionen
- ▶ Wenige Basisprimitive, darauf aufbauend Abstraktionen

6

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 ()

7

Rahmenbedingungen

- ▶ Zeitscheiben:
 - ▶ Tick: Default 20ms
 - ▶ Contextswitch pro Tick bei Heapallokation
 - ▶ Änderungen per Kommandozeilenoptionen: +RTS -V_itime_i -C_itime_i
- ▶ Blockierung:
 - ▶ Systemaufrufe blockieren alle Threads
 - ▶ Mit threaded library (-threaded) nicht alle
 - ▶ Aber: Haskell Standard-IO blockiert nur den aufrufenden Thread

8

Concurrent Haskell — erste Schritte

- ▶ Ein einfaches Beispiel:

```
write :: Char → IO ()
write c = putChar c >> write c

main :: IO ()
main = forkIO (write 'X') >> write 'O'
```

- ▶ Ausgabe ghc: (X*|O*)*
- ▶ Ausgabe hugs: (X*|O*)

9

Synchronisation mit MVars

- ▶ **Basissynchronisationsmechanismus** in Concurrent Haskell
 - ▶ Alles andere abgeleitet
- ▶ MVar *a* veränderbare Variable (vgl. IORef *a*)
- ▶ Entweder **leer** oder **gefüllt** mit Wert vom Typ *a*
- ▶ Verhalten beim Lesen und Schreiben

Zustand vorher:	leer	gefüllt
Lesen	blockiert (bis gefüllt)	danach leer
Schreiben	danach gefüllt	blockiert (bis leer)

- ▶ NB. Aufwecken blockierter Prozesse einzeln in FIFO

10

Basisfunktionen MVars

- ▶ Neue Variable erzeugen (leer oder gefüllt):

```
newEmptyMVar :: IO (MVar a)
newMVar :: a → IO (MVar a)
```

- ▶ Lesen:

```
takeMVar :: MVar a → IO a
```

- ▶ Schreiben:

```
putMVar :: MVar a → a → IO ()
```

11

Abgeleitete Funktionen MVars

- ▶ Nicht-blockierendes Lesen/Schreiben:

```
tryTakeMVar :: MVar a → IO (Maybe a)
tryPutMVar :: MVar a → a → IO Bool
```

- ▶ Änderung der MVar:

```
swapMVar :: MVar a → a → IO a
withMVar :: MVar a → (a → IO b) → IO b
modifyMVar :: MVar a → (a → IO (a, b)) → IO b
```

- ▶ **Achtung:** race conditions

12

Ein einfaches Beispiel ohne Synchronisation

- ▶ Nebenläufige Eingabe von der Tastatur

```
import Control.Monad(forever, replicateM_)
import Control.Concurrent

echo :: String → IO ()
echo p = forever $ do
  putStrLn ("***_Please_enter_line_for_" ++ p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n $ putStr (p ++ ":" ++ line ++ "\n")

main :: IO ()
main = forkIO (echo "2") >> echo "1"
```

- ▶ **Problem:** gleichzeitige Eingabe
- ▶ **Lösung:** MVar synchronisiert Eingabe

13

Ein einfaches Beispiel mit Synchronisation

- ▶ MVar voll ⇔ Eingabe möglich
 - ▶ Also: initial voll
- ▶ Inhalt der MVar irrelevant: MVar ()

```
echo :: MVar () → String → IO ()
echo flag p = forever $ do
  takeMVar flag
  putStrLn ("***_Please_enter_line_" ++ p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n $ putStr (p ++ ":" ++ line ++ "\n")
  putMVar flag ()

main :: IO ()
main = do flag ← newMVar ()
  forkIO (echo flag "3") >> forkIO (echo flag "2")
  echo flag "1"
```

14

Das Standardbeispiel

- ▶ Speisende Philosophen
- ▶ Philosoph *i*:
 - ▶ vor dem Essen *i*-tes und $(i+1)$ mod *n*-tes Stäbchen nehmen
 - ▶ nach dem Essen wieder zurücklegen
- ▶ Stäbchen modelliert als MVar ()

15

Speisende Philosophen

```
philos :: [MVar ()] → Int → IO ()
philos chopsticks i = forever $ do
  let num_phil = length chopsticks
      — Thinking:
      putStrLn ("Phil_#" ++ show i ++ "_thinks...")
      randomRIO (10, 200) >>= threadDelay
      — Get ready to eat:
      takeMVar (chopsticks !! i)
      takeMVar (chopsticks !! ((i+1) 'mod' num_phil))
      — Eat:
      putStrLn ("Phil_#" ++ show i ++ "_eats...")
      randomRIO (10, 200) >>= threadDelay
      — Done eating:
      putMVar (chopsticks !! i) ()
      putMVar (chopsticks !! ((i+1) 'mod' num_phil)) ()
```

16

Speisende Philosophen

- ▶ Hauptfunktion: n Stäbchen erzeugen
- ▶ Anzahl Philosophen in der Kommandozeile

```
main = do
  a:_ ← getArgs
  let num= read a
      chopsticks ← replicateM num $ newMVar ()
      mapM_ (forkIO . (philo chopsticks)) [0.. num-1]
  block
```

- ▶ Hilfsfunktion block: blockiert aufrufenden Thread

```
block :: IO ()
block = newEmptyMVar >>= takeMVar
```

- ▶ NB: Hauptthread terminiert — Programm terminiert!

17

Abstraktion: Semaphoren

- ▶ Abstrakter Datentyp QSem
- ▶ **Betret**en kritischer Abschnitt (P): `waitQSem :: QSem → IO ()`
- ▶ **Verlassen** kritischer Abschnitt (V): `signalQSem :: QSem → IO ()`
- ▶ Semaphore: Zähler plus evtl. wartende Threads
 - ▶ P erniedrigt Zähler, blockiert ggf. aufrufenden Thread
 - ▶ V erhöht Zähler, gibt ggf. blockierte Threads frei
- ▶ Implementierung von Semaphoren mit MVar: eigenes Scheduling
- ▶ Variation: **Quantitative Semaphoren**
 - ▶ Zähler kann um Parameter n erhöht/erniedrigt werden

18

Semaphoren: die P-Operation

```
data QSem = QSem (MVar (Int, [MVar ()]))
```

- ▶ MVar .. für die ganze Semaphore, darin:
- ▶ Zähler der Prozesse im kritischen Abschnitt
- ▶ Liste von wartenden Prozessen (MVar ())

```
newQSem :: Int → IO QSem
newQSem n = do m ← newMVar (n, [])
              return (QSem m)
```

19

Semaphoren: die P-Operation

- ▶ **Eintritt** in kritischen Abschnitt
- ▶ Wenn Eintritt möglich, Zähler erniedrigen
- ▶ Ansonsten blockieren (**Reihenfolge!**)

```
waitQSem :: QSem → IO ()
waitQSem (QSem sem) = do
  (avail, blocked) ← takeMVar sem
  if avail > 0 then
    putMVar sem (avail - 1, [])
  else do
    block ← newEmptyMVar
    putMVar sem (0, blocked ++ [block])
    takeMVar block
```

20

Semaphoren: die V-Operation

- ▶ **Verlassen** des kritischen Abschnitts
- ▶ Falls wartende threads, einen aufwecken.
- ▶ Alternatives Scheduling:
 - ▶ am Anfang hinzufügen, vom Anfang nehmen (**einfacher, unfair**)
 - ▶ am besten: **zufällige** Auswahl

```
signalQSem :: QSem → IO ()
signalQSem (QSem sem) = do
  (avail, blocked) ← takeMVar sem
  case blocked of
    [] → putMVar sem (avail + 1, [])
    block:blocked' → do
      putMVar sem (0, blocked')
      putMVar block ()
```

21

Zusammenfassung

- ▶ **Concurrent Haskell** bietet
 - ▶ **Threads** auf Quellsprachenebene
 - ▶ Synchronisierung mit MVars
 - ▶ Durch **schlankes Design** einfache Implementierung
- ▶ Funktionales Paradigma erlaubt **Abstraktionen**
 - ▶ Beispiel: Semaphoren
- ▶ Nächste Woche: Kanäle und Ausnahmen.

22