Nebenläufige Programmierung in Haskell: Grundlagen

Heute gibt's hier: Nebenläufigkeit

- Grundkonzepte
- Implementation in Haskell
- Basiskonzepte

Konzepte der Nebenläufigkeit

Thread (lightweight process)	vs.	Prozess
Programmiersprache		Betriebssystem
(z.B. Java, Haskell)		
gemeinsamer Speicher		getrennter Speicher
mehrere pro Programm		einer pro Programm

Multitasking:

- präemptiv: Kontextwechsel wird erzwungen
- kooperativ: Kontextwechsel nur freiwillig

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`
  - Fehlerhaft implementiert/spezifiziert?

Threads in Haskell: Concurrent Haskell

- Sequentielles Haskell: Reduktion eines Ausdrucks
- Nebenläufiges Haskell: Reduktion eines Ausdrucks an mehreren Stellen
- `ghc` (und `hugs`) implementieren Haskell-Threads
- `ghc`: präemptiv, `hugs`: kooperativ
- Modul `Concurrent` enthält Basisfunktionen
- Wenige Basisprimitive, darauf aufbauend Abstraktionen

Concurrent Haskell

- Jeder Thread hat einen Identifier — `type 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 ()`
  - Argument in Mikrosekunden
  - Auflösung $\cong 50ms$
- Blockierung:
  - Blockierende Systemaufrufe blockieren alle Threads
  - Aber: Haskell Standard-IO blockiert nur den aufrufenden Thread

Concurrent Haskell — erste Schritte

Ein einfaches Beispiel:

```
import Concurrent

write :: Char-> IO ()
write c = putChar c >> write c

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

Übersetzen: `ghc -package concurrent simple1.lhs`

Mit Hugs keine besonderen Optionen nötig.

Zeigen.

Synchronisation mit MVars

- Basisynchronisationmechanismus in `Concurrent Haskell`
  - Alles andere abgeleitet.
- `MVar a` veränderbare Variable (ähnlich `IORef a`)
- Entweder leer oder gefüllt mit einem `a`
- Verhalten beim Lesen und Schreiben

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

- 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 ()
```

- Nicht-blockierendes Lesen/Schreiben:

```
tryTakeMVar :: MVar a -> IO (Maybe a)
tryPutMVar  :: MVar a-> a-> IO Bool
```

- Test (Achtung: Zustand kann sich ändern)

```
isEmptyMVar :: MVar a -> IO Bool
```

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
echo :: String-> IO ()
echo p =
  do putStrLn ("\nPlease enter line for "++p)
     line <- getLine
     randomRIO (1,100) >>= \n-> forN n (putStr (p++ ":"++ line++" "))
     echo p
main :: IO ()
main = forkIO (echo "1") >> echo "2"
```

- Problem: gleichzeitige Eingabe Zeigen
- Lösung: `MVar` synchronisiert Eingabe

Ein einfaches Beispiel mit Synchronisation

- `MVar` voll $\Leftrightarrow$ Eingabe möglich
  - Also: initial voll
- Inhalt der MVar irrelevant: `MVar ()`

```
echo :: MVar ()-> String-> IO ()
echo flag p =
  do takeMVar flag
     putStrLn ("\nPlease enter line "++p)
     line <- getLine
     putMVar flag ()
     randomRIO (1,100) >>= \n-> forN n (putStr (p++ ":"++ line++" "))
     -- yield  -- noetig, da sonst keine Fairness.
     echo flag p
```

Das Standardbeispiel: Die speisenden Philosophen

- Philosoph i:
  - vor dem Essen i-tes und $(i+1) \bmod n$-tes Stäbchen nehmen
  - nach dem Essen wieder zurücklegen
- Stäbchen modelliert als `MVar ()`

- i-ter Philosoph:

```
philo :: [MVar ()] -> Int-> IO ()
philo chopsticks i =
   let num_phil = length (chopsticks)
   in do putStrLn ("Phil #"++(show i)++" thinks...")
         randomRIO (500,2000) >>= threadDelay
         takeMVar (chopsticks !! i)
         takeMVar (chopsticks !! ((i+1) `mod` num_phil))
         putStrLn ("Phil #"++(show i)++" eats...")
         randomRIO (500,2000) >>= threadDelay
         putMVar (chopsticks !! i) ()
         putMVar (chopsticks !! ((i+1) `mod` num_phil)) ()
         philo chopsticks i
```

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

```
main :: IO ()
main = do num <- getArgs >>= \a-> return (read (head a))
          chopsticks <- sequence
                          (take num (repeat (newMVar ())))
          mapM_ (forkIO . (philo chopsticks)) [0.. num-1]
          block
```

- Hilfsfunktion `block`: blockiert aufrufenden Thread

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

- NB: Hauptthread terminiert — Programm terminiert! Zeigen.

Zusammenfassung

Concurrent Haskell bietet

- Threads auf Quellsprachenebene
- Synchronisierung mit `MVars`
- Durch schlankes Design einfache Implementierung
- Funktionales Paradigma erlaubt Abstraktionen (nächste Woche!)