fhwedel
UNIVERSITY OF APPLIED SCIENCES

Concurrent Haskell

Markus Knofe, Alexander Treptow

09.12.2008

Agenda

Concurrency

Concurrent Haskell

Concurrency

Markus Knofe,
Alexander Treptow

Definition

Definition

Concurrency, bzw. Nebenläufigkeit, ist eine Eigenschaft von Systemen in denen mehrere berechnende Prozesse zur selben Zeit ausgeführt werden, die miteinander interagieren.

Viele unterschiedliche Systeme verfügen über Nebenläufigkeit, von
eng gekoppelten, synchronen, parallelen Systemen
bis
lose gekoppelten, asynchronen, verteilten Systemen.

Markus Knofe,
Alexander Treptow

Konzepte

Arten der Nebenläufigkeit

- Prozesse
  - getrennter Speicher (Kontextwechsel teuer)
  - einer pro Programm
  - im Kernelspace (Erstellen teuer)
- Threads
  - gemeinsamer Speicher (Kontextwechsel günstig)
  - mehrere pro Programm
  - üblicherweise im Userspace (Erstellen günstig)

Multitasking Arten

- kooperativ (freiwillige Kontrollrückgabe)
- präemptiv (Kern verteilt feste Zeitschlitze zum Rechnen)

Markus Knofe,
Alexander Treptow

Concurrency in Java

- Erweiterung von `Thread` oder `Runnable`
- durch Aufruf von `start()` wird `run()` aufgerufen
- Kontextwechsel mit `yield()`
- je nach JVM kooperativ oder präemptiv
- Synchronisation mit `synchronize`

Problem:

- Java Threads zu teuer (OS-abhängig)

Markus Knofe,
Alexander Treptow

Concurrent Haskell

Markus Knofe,
Alexander Treptow

Concurrency in Haskell

Nebenläufigkeit in Haskell mit `Control.Concurrent` Paket

- OS-Threads wie in Java möglich (`forkOS`)
- leichtgewichtige Threads (`forkIO`)
  - Kontextwechsel extrem günstig
  - Erstellen extrem günstig
  - verwendet keine OS gestützten Bibliotheken
- Kommunikation und Synchronisation über mutable Variables (`MVar`)
- nur GHC bietet vollen Paketumfang
- Hugs Threads arbeiten kooperativ

GHC vs. Hugs

2 Output-Threads

```
main = forkIO (write (take 20 (repeat 'a'))) >>
       write (take 20 (repeat 'b'))
  where
    write [] = putChar 'e'
    write (c:cs) = putChar c >> write cs
```

Ausgabe (Hugs):

```
> bbbbbbbbbbbbbbbbbbbeaaaaaaaaaaaaaaaaaaae
```

Ausgabe (GHCi):

```
> bababababababababababababababababababaee
```

Threads

- Unterteilung in System und Haskell Threads
- verfügen immer über eine `ThreadId`
- Haskell Threads können an Systemthreads gebunden werden
- `-threaded` Option
  - Haskell Runtime verfügt über mehrere OS-Threads
  - erlaubt Bound Threads
  - notwendig für `forkOS`
  - nötig für nicht blockierende `forkIO` Thread-Kommunikation
- System.IO blockiert nie (internes Multiplexing)

forkOS vs. forkIO

- `forkOS`
  - Zugriff auf Status des Threads (OpenGL)
  - Kontextswitch teuer
  - erstellt über Systemabhängige Pakete z.b. pthread
- `forkIO`
  - leichtgewichtiger Thread
  - günstig bzgl. Laufzeit und Speicher
  - vollständig in Haskell Runtime verwaltet

Zeiten für das Erzeugen von Threads

Threads (#)	forkIO (sek.)	forkOS (sek.)
5.000	0,02	0,31
10.000	0,03	0,62
50.000	0,19	3,12
100.000	0,36	6,16

Bound Threads

- immer fest an einen Systemthread gebunden
- werden mit `forkOS` erzeugt
- je nach OS wird *pthread_create*, *CreateThread*, etc. ausgeführt
- *thread-local state* nur über `forkOS`-Threads zugreifbar (OpenGL)
- `Main.main` ist immer ein Bound Thread

Bound Thread Funktionen

- `rtsSupportsBoundThreads :: Bool`
- `forkOS :: IO () -> IO ThreadId`
- `isCurrentThreadBound :: IO Bool`
- `runInBoundThread :: IO a -> IO a`
- `runInUnboundThread :: IO a -> IO a`

Erstellen eines Bound Thread

```
main = if rtsSupportsBoundThreads then
         forkOS (putChar 'x')
       else
         forkIO (putChar 'x')
```

Haskell Threads

- Konstruktor: `forkIO`
- leichtgewichtige Threads
- komplett von der Haskell Runtime verwaltet
- effizient in Laufzeit und Speicher
- Problem: Was wenn ein Thread blockiert?
- Lösung: mehrere OS-Threads in der Haskell Runtime
- ohne `-threaded` Option, keine *worker*-Threads
- I/O ist nie blockierend, da `System.IO` multiplext

Markus Knofe,
Alexander Treptow

forkIO

- `forkIO :: IO () -> IO ThreadId`
- startet einen neuen leichtgewichtigen Thread, der die übergebene IO-Berechnung ausführt
- gibt ein Handle für den Thread zurück
- automatisches Exceptionhandling für
  `BlockedOnDeadMVar, BlockedIndefinitely ThreadKilled`
- ansonsten `setUncaughtExceptionHandler`

Beispiel

```
main = forkIO (write 'a') >> write 'b'
       where
       write c = putChar c >> write c
```


Markus Knofe,
Alexander Treptow

Control.Concurrent Interface

- `myThreadId :: IO ThreadId`
- `killThread :: ThreadId -> IO ()`

  `killThread tid = throwTo tid ThreadKilled`
- `throwTo :: Exception e => ThreadId -> e -> IO ()`
- `yield :: IO ()`

  wie in Java, Kontrolle an anderen Thread geben
- `threadDelay :: Int -> IO ()`
- `threadWaitRead :: Fd -> IO ()`
- `threadWaitWrite :: Fd -> IO ()`
- `mergeIO :: [a] -> [a] -> IO [a]`
- `nmergeIO :: [[a]] -> IO [a]`

  `mergeIO` und `nmergeIO` sind nur präemptiv möglich

fhwedel
UNIVERSITY OF APPLIED SCIENCES

Thread Kommunikation

"[...] based on the idea that two processes could communicate via a lazy-evaluated list, produced by one and consumed by the other. [...]"

(Aus "Concurrent Haskell" von Simon Peyton Jones, Andew Gordon & Sigbjorn Finne)

Markus Knofe,
Alexander Treptow

Ein Consumer Beispiel

Ein Beispiel: Kommunikation über eine lazy-evaluated-List

Ein Consumer

```
main =  producer >> waitForQ
producer = do
  l <- return ( mkInfList )
  forkIO $ consumer l
  where
    consumer (x:xs) = hPutChar stdout 'c' >>
   hPrint stdout (show x) >> consumer xs
    mkInfList :: [Int]
    mkInfList   = [] ++ (genList 1)
    where
      genList i = [i] ++ genList (i+1)
```


Markus Knofe,
Alexander Treptow

Zwei Consumer Beispiel

Nun erweitern wir das Beispiel auf zwei Consumer

Zwei Consumer

```
producer = do
  l <- return ( mkInfList )
  forkIO $ consumer "odd: " l
  forkIO $ consumer "even:" l
  where
    consumer name (x:xs) = hPutStr stdout name
    >> hPrint stdout (show x) >> consumer name
    xs
    mkInfList :: [Int]
    mkInfList   = [] ++ (genList 1)
          where
            genList i = [i] ++ genList (i+1)
```

Aufspalten der Liste

Anstelle von einer Liste einfach zwei Listen

Zwei Consumer, zwei Listen

```
main =  forkIO( printnumber_proc "odd"  [1,3..]
     ) >> forkIO( printnumber_proc "even"
    [2,4..] ) >> waitForQ
  where
    printnumber_proc name (x:xs) = hPutStr
    stdout name >> hPutStrLn stdout (show x) >>
     printnumber_proc name xs
```

Probleme:

- Mit mehreren Threads auf einer Datei zu arbeiten
- Einen Stream lesen, der von mehr als einem Thread beschrieben wird

MVar

Lösung: `MVar`

- Dient der Syncronisation
- Definiert durch "`type MVar a`"
- Ist Typen gebunden
- Hat zwei Zustände
  - leer
  - belegt mit einem Wert des gebundenen Typen

Markus Knofe,
Alexander Treptow

Interface

```
newMVar :: IO (MVar a)
```
Erzeugt eine neue `MVar`

```
takeMVar :: MVar a -> IO a
```
`MVar` Leer $\Rightarrow$ blockiert solange die `MVar` leer ist
`MVar` Voll $\Rightarrow$ leert die `MVar` und liefert deren vorherigen Wert

```
putMVar :: MVar a -> a -> IO ()
```
`MVar` Leer $\Rightarrow$ schreibt Wert in die `MVar` und weckt eventuelle "take"-Threads in FIFO Reihenfolge
`MVar` Voll $\Rightarrow$ blockiert solange die `MVar` voll ist

Anwendungen:

- als einfacher Semaphor
- zur Syncronisation von z.B. IO-Zugriffen

Synchonisation

Synchronisation der Ausgabe mittels einer `MVar`

```
main =  do
  mvar <- newEmptyMVar  :: IO ( MVar Bool )
  forkIO( printnumber_proc "odd" mvar [1,3..])
  forkIO( printnumber_proc "even" mvar [2,4..])
  waitForQ
  where
    printnumber_proc name mvar (x:xs) = putMVar
     mvar True >> hPutStr stdout name >>
    hPutStrLn stdout (show x) >> takeMVar mvar
    >> printnumber_proc name mvar xs
```

Verbesserung der Synchronisation 1

Verbesserung durch zweite `MVar` 1

```
type EOVar = (MVar (), MVar ())
newEOVar = newEmptyMVar >>= \ odd_var ->
    newEmptyMVar >>= \ even_var -> putMVar
    even_var () >> return (odd_var, even_var)
aroundE (odd_var,even_var) f s x = takeMVar
    odd_var  >> f s x >> putMVar even_var ()
aroundO (odd_var,even_var) f s x = takeMVar
    even_var >> f s x >> putMVar odd_var ()
```

[...]

Verbesserung der Synchronisation 2

Verbesserung durch zweite `MVar` 2

[...]

```
main = do
  eovar  <- newEOVar
  forkIO( print_odd eovar [1,3..] )
  forkIO( print_even eovar [2,4..] )
  waitForQ
  where
    print_odd eovar (x:xs) =  aroundO eovar
    printX "odd:  " x >> print_odd eovar xs
    print_even eovar (x:xs) = aroundE eovar
    printX "even: " x >> print_even eovar xs
    printX n x = hPutStr stdout (n ++ (show x)
    ++ "\n")
```

MVar zu Chan

Gut: die Ausgabe und Reihenfolgen stimmen

Schlecht: jeder Thread hat eigene Datenquelle

Lösung: Channel, synchronisiert das Schreiben und Lesen auf einen Stream

Markus Knofe,
Alexander Treptow

Chan Interface

```
type Channel a = (MVar (Stream a), MVar (Stream a))
type Stream a = MVar (Item a)
Data Item a = Item a (Stream a)

newChan :: IO (Channel a)
```

Erzeugt einen neuen Channel

```
writeChan :: Channel a -> a ->IO ()
```

Schreibt einen Wert in den Channel

```
getChanContents :: Channel a -> IO [a]
```

Liefert eine Repräsentation des Channelinhalts als "lazy-evaluated" List

Even-Odd mit Chan 1

Even-Odd Beispiel mit einer Chan 1

```
main =  do
  chan           <- newChan :: IO Pipe
  llist          <- getChanContents chan
  oddThread      <- forkIO ( printnumber_proc "
    odd:  " (catOdd  llist))
  evenThread     <- forkIO ( printnumber_proc "
    even: " (catEven llist))
  numbersThread <- forkIO ( numbers chan [1..])
  waitForQ
  where


[...]
```

Even-Odd mit Chan 2

Even-Odd Beispiel mit einer Chan 2

```
[...]

    printnumber_proc name (x:xs) = [...]
    catOdd  ls = [x | Left  x <- ls]
    catEven ls = [x | Right x <- ls]

numbers chan (x:xs) =
  if odd x
  then writeChan chan (Left  x) >> threadDelay
    someTime >> numbers chan xs
  else writeChan chan (Right  x) >> threadDelay
     someTime >> numbers chan xs
```

Was ist waitForQ?

Problem:

- Ein GHC compiliertes Programm läuft nur solange wie sein Hauptthread
  $\Rightarrow$ Sobald der Hauptthread beendet ist werden auch alle Kindthreads beendet

Einfache Lösung:

- Den Hauptthread mittels einer Schleife am Leben halten (`waitForQ`)

Bessere Lösung:

- Das Beenden der Kindthreads abwarten und dann den Haupthread beenden

waitForQ mit MVar 1

Realisierung mittels `MVar` 1

```
children :: MVar [MVar ()]
children = unsafePerformIO (newMVar [])

waitForChildren :: IO ()
waitForChildren = do
  cs <- takeMVar children
  case cs of
    []   -> return ()
    m:ms -> do
      putMVar children ms
      takeMVar m
      waitForChildren
```

[...]

Markus Knofe,
Alexander Treptow

waitForQ mit MVar 2

Realisierung mittels `MVar` 2

[...]

```
forkChild :: IO () -> IO ThreadId
forkChild io = do
  mvar <- newEmptyMVar
  childs <- takeMVar children
  putMVar children (mvar:childs)
  forkIO (io `finally` putMVar mvar ())
```

Fakultät berechnen

Praktische Anwendung: Fakultät berechnen mit MapAndReduce

Markus Knofe,
Alexander Treptow

Noch Fragen?

Markus Knofe,
Alexander Treptow