Concurrent Haskell

Seminararbeit im SS 2003

Volker Hofmann

Überblick

Vortragsgliederung

1. Einleitung
2. Grundlagen
3. Standardabstraktionen
4. Scheduling
5. Implementierung
6. Schlussworte

Einleitung

Was ist Concurrent Haskell?

- …führt das Konzept der **Nebenläufigkeit** ein
- …ist eine **Erweiterung** der funktionalen Sprache Haskell
- …besteht aus nur **4 zusätzlichen primitiven Operationen**

Einleitung

Auf dieser Grundlage wird entwickelt:

- Puffer
- Kanal
- Steuerung des Scheduling
- …

Grundlage ist überraschend flexibel und ausdrucksstark!

Überblick

Vortragsgliederung

1. Einleitung
2. Grundlagen
3. Standardabstraktionen
4. Scheduling
5. Implementierung
6. Schlussworte

Grundlagen

Erweiterung von Haskell um zwei wesentliche Inhalte:

- **Prozesse** und Mechanismen zum Erzeugen von Prozessen
- **Interprozesskommunikation** mit Hilfe von atomar-veränderbaren Variablen

Grundlagen > Konzept der EA-Monads

- Eigenschaft von Haskell: **nicht-strikt**
- Ungeeignet für EA-Operationen

Grundlagen > EA Monads

- Lösung:

  Behandeln von EA-Operationen als Zustandsveränderungen
- Durch Monads wird sequenzialisieren und zusammenfassen der Operationen möglich

Grundlagen > EA Monads

- Typdefinition:

  type IO a = World $\rightarrow$ (a, World)

- Beispiel:

```
hGetChar :: Handle -> IO Char
hPutChar :: Handle -> Char -> IO ()
```

Fortführung des Beispiels:

```
hGetChar stdin >>= \c -> hPutChar stdout c
```

Oder:

```
do
   c <- hGetChar stdin
   hPutChar stdout c
```

Grundlagen > EA Monads

Einsatz von IO in Interprozesskommunikation:

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

Grundlagen > EA Monads

- Konzept der EA Monads ist **wichtig**
- Sogar Programme liefern Typ `IO ()`

Neue Operation:

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

Startet einen neuen konkurrierenden Prozess

Beispiel:

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

Produziert z.B.:

bbabaabbbabaabbaaaababbab

Wichtig:

Es wird **Interprozesskommunikation** erforderlich

Einführung von Nicht-Determinismus. Sichere Veränderung der „Welt“ durch Einsatz von EA-Monads.

Eine Möglichkeit:

- Synchronisation durch Haskell-Implementierung
- Kommunikation über Streams

Gründe für die Einführung weiterer Mechanismen zur Synchronisation und Kommunikation:

- Betriebsmittel müssen **exklusiv** verfügbar sein
- **Stream-Programmierung sehr mühevoll**

Lösung: Einführung eines neuen primitiven Typs (geteilte, veränderbare Sperrvariable):

```
type MVar a
```

Operationen:

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

Sichtweise: Für einen Prozess sind alle anderen Prozesse Teil des „Rest der Welts“

MVar kann angesehen werden als:

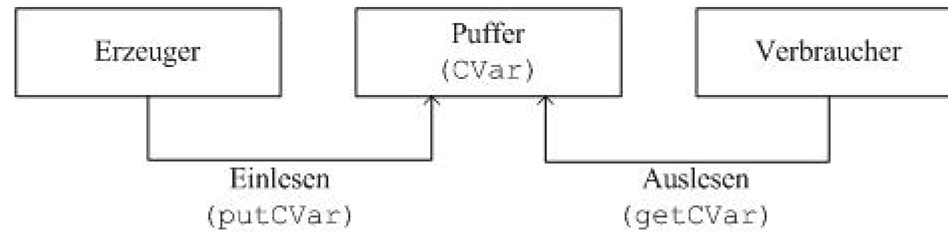
- Binäre Semaphore mit Wait- und Signal-Operationen repräsentiert durch `takeMVar` und `putMVar`
- Kommunikationskanal mit `takeMVar` und `putMVar` zum Senden und Empfangen

Überblick

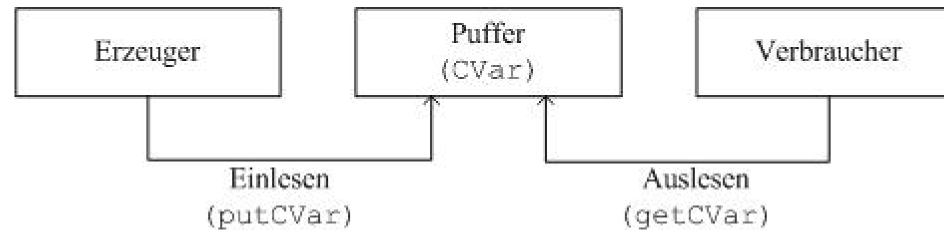
Vortragsgliederung

1. Einleitung
2. Grundlagen
3. **Standardabstraktionen**
4. Spezialitäten: Scheduling, Prozessauswahl
5. Implementierung
6. Schlussworte

Standardabstraktionen > Puffer

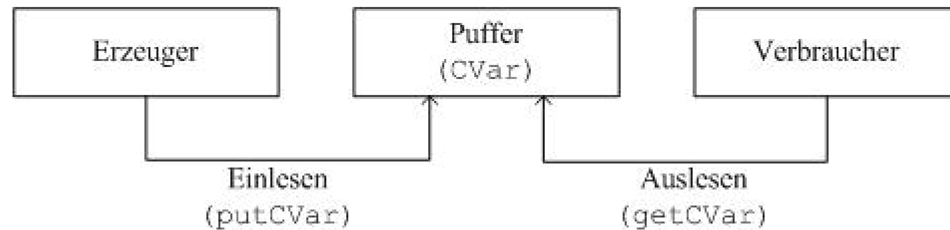


Standardabstraktionen > Puffer




```
type CVar a = (MVar a,  -- Produzent -> Konsument
               MVar ()) -- Konsument -> Produzent

newCVar :: IO (CVar a)
newCVar
  = newMVar              >>= \ data_var ->
    newMVar              >>= \ ack_var  ->
    putMVar ack_var ()  >>
    return (data_var,ack_var)

getCVar :: CVar a -> IO a
getCVar (data_var,ack_var)
  = takeMVar data_var  >> = \ val ->
    putMVar ack_var () >>

    return val

putCVar :: CVar a -> a -> IO ()
putCVar (data_var,ack_var) val
  = takeMVar ack_var >>
    putMVar data_var val
```

Standardabstraktionen > Puffer




```
type CVar a = (MVar a,  -- Produzent -> Konsument
               MVar ()) -- Konsument -> Produzent

newCVar :: IO (CVar a)
newCVar
  = newMVar             >>= \ data_var ->
    newMVar             >>= \ ack_var  ->
    putMVar ack_var ()  >>
    return (data_var,ack_var)

getCVar :: CVar a -> a -> IO a
getCVar (data_var,ack_var)
  = takeMVar data_var  >> = \ val ->
    putMVar ack_var () >>

    return val

putCVar :: CVar a -> a -> IO ()
putCVar (data_var,ack_var) val
  = takeMVar ack_var >>
    putMVar data_var val
```

Weitere Standardabstraktionen:

- Unbegrenzter Puffer
- …

Überblick

Vortragsgliederung

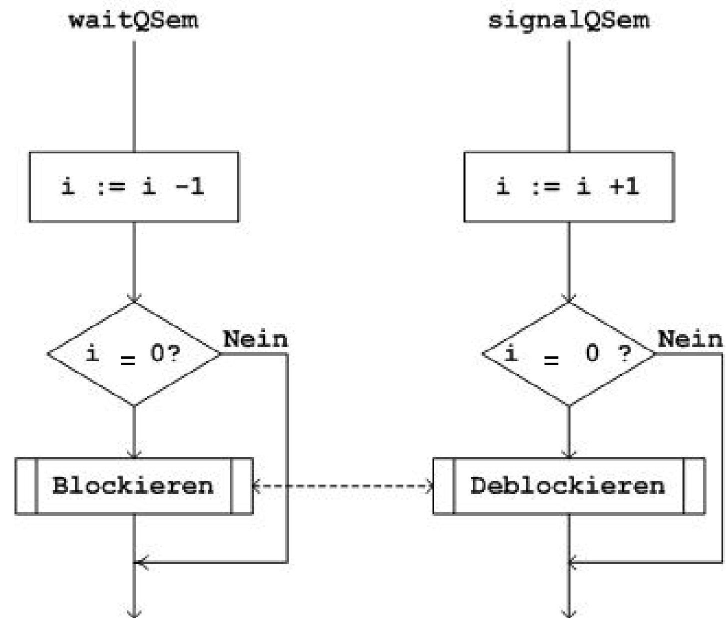
1. Einleitung
2. Grundlagen
3. Standardabstraktionen
4. **Scheduling**
5. Implementierung
6. Schlussworte

Scheduling

Kontrolle über Scheduling durch `MVar`:

- Leere `MVar` repräsentiert blockierten Prozess
- Steuerung des Prozesses über Schreiben in `MVar`

Scheduling > Grundideen




```
type QSem = MVar (Int, [MVar ()])
newQSem :: IO QSem
waitQSem :: QSem -> IO ()
signalQSem :: QSem -> IO ()
```

Scheduling > Implementierung

```
waitQSem sem
    = takeMVar sem >>= \(avail,blocked) ->
    if avail > 0 then
         putMVar (avail-1, []) >>
    else
         newMVar >>=\block ->
         putMVar (0, block:blkd) >>
         takeMVar block

signalQSem sem
    = takeMVar sem >>=\(avail,blkd) ->
         case blkd of
         [] -> putMVar (avail+1, [])
         (blocked:blockedTail) -> putMVar (0, blockedTail) >>
            putMVar blocked ()
```

- Concurrent Haskell bietet keine prioritätsgesteuerten Prozessauswahlstrategien an

  → eigene Implementierung, z.B. :

  Liste mit blockierten Prozessen und Prioritäten

Überblick

Vortragsgliederung

1. Einleitung
2. Grundlagen
3. Standardabstraktionen
4. Scheduling
5. **Implementierung**
6. Schlussworte

Implementierung

Prozesse:

- Concurrent Haskell ist ein Prozess im Betriebssystem
- `forkIO` arbeitet innerhalb dieses Prozesses

Implementierung

Implementierung der `MVar`:

- Zeiger auf Heap-Bereich
- Im Heap Bereich:

  Wert
  Flag (leer/voll?)
  Liste der wartenden Prozesse

Überblick

Vortragsgliederung

1. Einleitung
2. Grundlagen
3. Standardabstraktionen
4. Scheduling
5. Implementierung
6. **Schlussworte**

Schlussworte

Concurrent Haskell ist:

- einfach
- sehr ausdrucksstark

Schlussworte

Concurrent Haskell gewinnt an Wert durch:

- Distributed Haskell
- Parallel Haskell

Vielen Dank!

Ende