# Concurrent Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik


Letzte Änderung der Folien: 30. Dezember 2020

## Übersicht





Concurrent Haskell

## Concurrent Haskell

- Erweiterung von Haskell um Nebenläufigkeit
- Integriert in das monadische IO
- Beachte: Das Welt-Modell passt nicht
- Besseres Modell: Operationale Semantik direkt angeben

## Concurrent Haskell (2)

- Neue Konstrukte in Bibliothek: Control.Concurrent
- zum Erzeugen nebenläufiger Threads:

  eine neue primitive Operation
- zur Kommunikation / Synchronisation von Threads

  ein neuer Datentyp mit drei primitiven Operationen



## Erzeugen eines nebenläufigen Threads

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

- `forkIO t` führt Aktion `t` in nebenläufigem Thread aus
- im Hauptthread: sofortiges Ergebnis = eindeutige ID
- Hauptthread läuft weiter
- Ende des Hauptthreads beendet alle nebenläufigen
- `killThread :: ThreadId -> IO ()`



## Beispiel: Zweimal Echo

```
echo i = do
 putStr $ "Eingabe fuer Thread" ++ show i ++ ":"
 line <- getLine
 putStrLn $ "Letzte Eingabe fuer Thread" ++ show i ++  ":" ++ line
 echo i

zweiEchos = do
             forkIO (echo 1)
             forkIO (echo 2)
             block  -- Hauptthread blockieren

block = do block
```



## Der Datentyp `MVar`

`MVar` (mutable variable) = Bounded Buffer der Größe 1

**Basisoperationen:**

- `newEmptyMVar :: IO (MVar a)`
  erzeugt leere `MVar`
- `takeMVar :: MVar a -> IO a`
  - liest Wert aus `MVar`, danach ist die `MVar` leer
  - falls `MVar` vorher leer: Thread wartet
  - Bei mehreren Threads: FIFO-Warteschlange
- `putMVar :: MVar a -> a -> IO ()`
  - speichert Wert in der `MVar`, wenn diese leer ist
  - Falls belegt: Thread wartet
  - Bei mehreren Threads: FIFO-Warteschlange

## Der Datentyp `MVar` (2)

| MVar | `takeMVar` |
|---|---|
| leer | warten |
| gefüllt | lesen |

| MVar | `putMVar` |
|---|---|
| leer | schreiben |
| gefüllt | warten |

Darstellung:

Leere MVar


Volle MVar



## MVars zur Synchronisation

Main-Thread wartet auf nebenläufige Threads:

```
main = do
        syncT1 <- newEmptyMVar
        syncT2 <- newEmptyMVar
        forkIO (thread1 >> putMVar syncT1 ())
        forkIO (thread2 >> putMVar syncT2 ())
        takeMVar syncT1
        takeMVar syncT2

thread1 = ...
thread2 = ...
```



## MVars zum Schützen von Daten

Zähler mit atomarer Operation zum Inkrementieren

```
type Counter = MVar Int

atomicIncrement m = do
  r <- takeMVar m
  putMVar m (r+1)
```



## MVars zum Schützen von Kritischen Abschnitten

```
takeMVar mutex
... kritischer Abschnitt ...
putMVar mutex ()
```

Dabei ist `mutex` eine mit () gefüllte `MVar` vom Typ `MVar ()`
Analog geht auch (wenn `mutex` am Anfang leer)

```
putMVar mutex ()
... kritischer Abschnitt ...
takeMVar mutex
```

## MVars als binärer Semaphor

```
type Semaphore = MVar ()

newSem     :: IO Semaphore
newSem     =  newEmptyMVar

wait       :: Semaphore -> IO ()
wait sem   =  takeMVar sem

signal     :: Semaphore -> IO ()
signal sem =  putMVar sem ()
```

- newSem "erzeugt Semaphor mit 0 initialisiert" (= leere MVar)
- `wait` blockiert leere MVar (also wenn k = 0)
- `signal` füllt MVar, wenn es blockierte Prozesse gibt, kann einer `wait` durchführen
- `signal` bei voller MVar blockiert (bei binären Semaphore: undefiniert)



## Schützen kritischer Abschnitte mit Semaphor

```
echoS sem i =
 do
  wait sem
  putStr $ "Eingabe fuer Thread" ++ show i ++ ":"
  line <- getLine
  signal sem
  wait sem
  putStrLn $ "Letzte Eingabe fuer Thread" ++ show i ++  ":" ++ line
  signal sem
  echoS sem i

zweiEchosS = do
              sem <- newSem
              signal sem
              forkIO (echoS sem 1)
              forkIO (echoS sem 2)
              block
```



## Weitere Bibliotheksfunktionen für MVar's

- `newMVar:: a -> IO (MVar a)`
  Erzeugt eine gefüllte MVar
- `readMVar :: MVar a -> IO a`
  Liest den Wert einer MVar (Kombination aus take und put)
- `swapMVar :: MVar a -> a -> IO a`
  Tauscht den Wert einer MVar aus (Race Condition möglich)
- `tryTakeMVar :: MVar a -> IO (Maybe a)`
  Nicht-blockierende Version von `takeMVar`
- `tryPutMVar :: MVar a -> a -> IO Bool`
  Nicht-blockierende Version von `putMVar`
- `isEmptyMVar :: MVar a -> IO Bool`
  Testet, ob `MVar` leer ist
- `modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()`
  Funktion anwenden auf den Inhalt einer `MVar`. Sicher bei Exceptions: Tritt ein Fehler auf, bleibt der alte Wert erhalten



## Weitere Funktionalitäten in Concurrent Haskell

- `forkIO :: IO () -> IO ThreadId`
  Erzeugt: Einen neuen "lightweight thread": Verwaltung durch das Laufzeitsytem
- `forkOS :: IO () -> IO ThreadId`
  Erzeugt einen "bound thread": Verwaltung durch das Betriebssystem
- `killThread :: ThreadId -> IO ()`
  beendet den Thread. Wenn Thread schon beendet, kein Effekt.
- `yield :: IO ()`
  Forciert einen Context-Switch: Der aktuelle Thread wird von aktiv auf bereit gesetzt. Ein anderer Thread darf Schritte machen.
- `threadDelay :: Int -> IO ()`
  Verzögert den aufrufenden Thread um die gegebene Zahl an Mikrosekunden.

## Erzeuger / Verbraucher mit 1-Platz Puffer

Erzeuger berechnet Werte und gibt sie an den Verbraucher

Synchronität: Erzeuger gibt erst den nächsten Wert, wenn Verbraucher den alten erhalten hat.

Puffer durch `MVar`: Lässt sich direkt implementieren

- `type Buffer a = MVar a`
- neuer Puffer: `newBuffer = newEmptyMVar`
- schreiben (Erzeuger): `writeToBuffer = putMVar`
- lesen (Verbraucher): `readFromBuffer = takeMVar`



## Erzeuger / Verbraucher mit 1-Platz Puffer (2)

```
newBuffer = newEmptyMVar
writeToBuffer = putMVar
readFromBuffer = takeMVar
```

```
                    buff <- newBuffer
                            ...
Prozess 1                          Prozess 2
do                                 do
 readFromBuffer buf                  writeToBuffer buf 1
                                     writeToBuffer buf 2
                                     writeToBuffer buf 3
```




## MVars zur Verwaltung von Zuständen

- Beispiel aus Buch von Simon Marlow
- Telefonbuch als `Map` von Name auf Telefonnummer
- Threads greifen auf das Telefonbuch zu
- Zustand wird in einer `MVar` gespeichert

```
type Name                = String
type Telefonnummer       = String
type Telefonbuch         = Map.Map Name Telefonnummer
type TelefonbuchZustand = MVar Telefonbuch
```

(`Map` ist eine Datenstruktur aus der Bibliothek `Data.Map`)



## MVars zur Verwaltung von Zuständen (2)

```
neu :: IO TelefonbuchZustand
neu = newMVar Map.empty

einfuegen :: TelefonbuchZustand -> Name -> Telefonnummer -> IO ()
einfuegen m name nummer = do
 buch <- takeMVar m
 putMVar m (Map.insert name nummer buch)

nachschauen :: TelefonbuchZustand -> Name -> IO (Maybe Telefonnummer)
nachschauen m name = do
 buch <- takeMVar m
 putMVar m buch
 return (Map.lookup name buch)

mainTel = do
 s <- neu
 sequence_ [einfuegen s ("name" ++ show i) (show i) | i <- [1..10000]]
 nachschauen s "name123" >>= print
 nachschauen s "unkown" >>= print
```

## Auswirkungen der call-by-need Auswertung

Beim Einfügen wird in

```
putMVar m (Map.insert name nummer buch)
```

der unausgewertete Ausdruck

```
(Map.insert name nummer buch)
```

in die `MVar` geschrieben. Kann zu einem space-leak führen!

**Lösungen:**

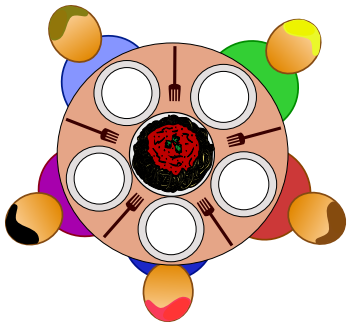
```
putMVar m $! (Map.insert name nummer buch)
```

noch besser:

```
einfuegen m name nummer = do
 buch <- takeMVar m
 let buch' = Map.insert name nummer buch
 putMVar m buch'
 seq buch' (return ())
```



## Speisende Philosophen



**Implementierung:**

- Eine Gabel durch eine `MVar ()`
- Ein Philosoph durch einen Thread



## Speisende Philosophen (2)

**1. Implementierung:** $i$-ter Philopsoph:

```
philosoph i gabeln =
 do
  let n = length gabeln                  -- Anzahl Gabeln
  takeMVar $ gabeln!!i                   -- nehme linke Gabel
  putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel ..."
  takeMVar $ gabeln!!(mod (i+1) n)       -- nehme rechte Gabel
  putStrLn $ "Philosoph " ++ show i ++ " isst ..."
  putMVar (gabeln!!i) ()                 -- lege linke Gabel ab
  putMVar (gabeln!!(mod (i+1) n)) ()     -- lege rechte Gabel ab
  putStrLn $ "Philosoph " ++ show i ++ " denkt ..."
  philosoph i gabeln
```



## Speisende Philosophen (3)

**1. Implementierung:** Erzeugen der Philosophen

```
philosophen n =
 do
  -- erzeuge Gabeln (n MVars):
  gabeln <- sequence $ replicate n (newMVar ())
  -- erzeuge Philosophen:
  sequence_ [forkIO (philosoph i gabeln) | i <- [0..n-1]]
  block
```

Beachte: Deadlock möglich!

## Speisende Philosophen (4)

**2. Implementierung:** $N$.-Philosoph in anderer Reihenfolge

```
philosophAsym i gabeln =
  do
   let n = length gabeln -- Anzahl Gabeln
   if length gabeln == i+1 then -- letzter Philosoph
     do takeMVar $ gabeln!!(mod (i+1) n) -- nehme rechte Gabel
        putStrLn $ "Philosoph " ++ show i ++ " hat rechte Gabel ..."
        takeMVar $ gabeln!!i  -- nehme linke Gabel
        putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel und isst"
    else
     do
      takeMVar $ gabeln!!i  -- nehme linke Gabel
      putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel ..."
      takeMVar $ gabeln!!(mod (i+1) n) -- nehme rechte Gabel
      putStrLn $ "Philosoph " ++ show i ++ " hat rechte Gabel und isst"
   putMVar (gabeln!!i) () -- lege linke Gabel ab
   putMVar (gabeln!!(mod (i+1) n)) () -- lege rechte Gabel ab
   philosophAsym i gabeln
```



## Generelle Semaphore

- Man kann generelle Semaphore mit binären Semaphore implementieren
- Aber: Kodierung ziemlich kompliziert
- Mit Haskells `MVars` jedoch leicht
- Implementierung in Bibliothek `Control.Concurrent.QSem`

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

- Das Paar `(Int,[MVar ()])` stellt die Komponenten des Semaphores dar: Zahl $k$ und Menge der wartenden Prozesse.
- Warten durch blockieren an `MVars`
- Die äußere `MVar` hält dieses Paar. Dadurch ist exklusiver Zugriff auf die Komponenten gesichert.



## Generelle Semaphore (2)

**Erzeugen einer** `QSem`

```
newQSem :: Int -> IO QSem
newQSem k = do
             sem <- newMVar (k, [])
             return sem
```



## Generelle Semaphore (3)

**wait-Operation**

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

## Generelle Semaphore (4)

**signal-Operation**

```
signalQSem :: QSem -> IO ()
signalQSem sem = do
   (k,blocked) <- takeMVar sem
   case blocked of
     [] -> putMVar sem (k+1,[])
     (block:blocked') -> do
           putMVar sem (0,blocked')
           putMVar block ()
```



## Speisende Philosophen 3. Implementierung

**3. Implementierung:** Mit Semaphor Raum

```
philosophRaum i raum gabeln = do
  let n = length gabeln -- Anzahl Gabeln
  waitQSem raum -- genereller Semaphor
  putStrLn $ "Philosoph " ++ show i ++ " im Raum"
  takeMVar $ gabeln!!i  -- nehme linke Gabel
  putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel ..."
  takeMVar $ gabeln!!(mod (i+1) n) -- nehme rechte Gabel
  putStrLn $ "Philosoph " ++ show i ++ " hat rechte Gabel und isst"
  putMVar (gabeln!!i) () -- lege linke Gabel ab
  putMVar (gabeln!!(mod (i+1) n)) () -- lege rechte Gabel ab
  signalQSem raum
  putStrLn $ "Philosoph " ++ show i ++ " aus Raum raus"
  putStrLn $ "Philosoph " ++ show i ++ " denkt ..."
  philosophRaum i raum gabeln

philosophenRaum n =
 do
  gabeln <- sequence $ replicate n (newMVar ())
  raum <- newQSem (n-1)
  sequence [forkIO (philosophRaum i raum gabeln) | i <- [0..n-1]]
  block
```



## Erweiterte generelle Semaphore

- `wait` und `signal` erhalten zusätzlich eine Zahl.
- Bei `wait` bedeutet die Zahl: Soviel "Ressourcen" müssen vorhanden sein, um entblockiert zu werden
- Bei `signal`: Soviel Ressourcen werden zu den vorhandenen hinzu gegeben.

```
type QSemN = MVar (Int,[(Int,MVar ())])
```

- In der Liste werden nun Paare gespeichert:
  - Zahl, wieviel Ressourcen noch benötigt werden
  - `MVar` zum blockieren.



## Erweiterte generelle Semaphore (2)

**Erzeugen:**

```
newQSemN :: Int -> IO QSemN
newQSemN initial = do
   sem <- newMVar (initial, [])
   return sem
```

## Erweiterte generelle Semaphore (3)

**Warten:**

```
waitQSemN :: QSemN -> Int -> IO ()
waitQSemN sem sz = do
  (k,blocked) <- takeMVar sem
  if (k - sz) >= 0 then
     putMVar sem (k-sz,blocked)
   else do
     block <- newEmptyMVar
     putMVar sem (k, blocked++[(sz,block)])
     takeMVar block
```



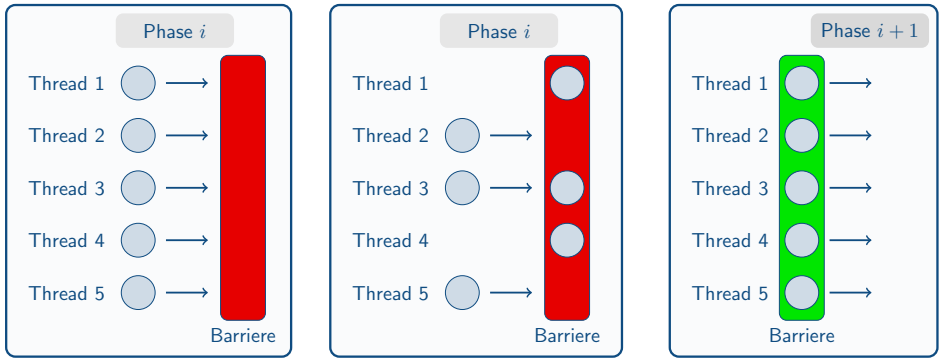
## Erweiterte generelle Semaphore (4)

**Signalisieren:**

```
signalQSemN :: QSemN -> Int  -> IO ()
signalQSemN sem n = do
   (k,blocked)   <- takeMVar sem
   (k',blocked') <- free (k+n) blocked
   putMVar sem (k',blocked')
 where
   free k []     = return (k,[])
   free k ((req,block):blocked)
     | k >= req = do
        putMVar block ()
        free (k-req) blocked
     | otherwise    = do
        (k',blocked') <- free k blocked
        return (k',(req,block):blocked')
```



## Barrierimplementierung mit `QSemN`






## Barrierimplementierung mit `QSemN` (2)

```
type Barrier = (Int, MVar (Int, QSemN))
```

- Erste Zahl: Anzahl der zu synchronisierenden Prozesse
- Zweite Zahl: Anzahl der aktuell angekommenen Prozesse
- QSemN: Wird benutzt um Prozesse zu blockieren
- MVar: Schützt Zugriff auf aktuelle Anzahl und QSemN

## Barrierimplementierung mit QSemN (3)

`newBarrier` erzeugt einen neuen Barrier

```
newBarrier n =
 do
  qsem <- newQSemN 0
  mvar <- newMVar (0,qsem)
  return (n,mvar)
```



## Barrierimplementierung mit QSemN (4)

Die Hauptfunktion ist `synchBarrier`

```
synchBarrier :: Barrier -> IO ()
synchBarrier (maxP,barrier) = do
 (angekommen,qsem) <- takeMVar barrier
 if angekommen+1 < maxP then do
   putMVar barrier (angekommen+1,qsem)
   waitQSemN  qsem 1
 else do
   signalQSemN qsem (maxP-1)
   putMVar barrier (0,qsem)
```



## Barrierimplementierung mit QSemN (5)

Code für die einzelnen Prozesse ist von der Form

```
prozess_i barrier = do
  -- Code fuer die aktuelle Phase ..
  synchBarrier barrier -- warte auf Synchronisierung
  prozess_i barrier -- starte n\"achste Phase
```



## Unbounded Buffer

FIFO-Kanal, an einem Ende schreiben, am anderen Ende lesen
Gewünschte Schnittstelle

- `type Kanal a`
- `neuerKanal :: IO (Kanal a)`
- `schreibe :: Kanal a -> a -> IO ()`
- `lese :: Kanal a -> IO a`

Außerdem: Keine Fehler, wenn mehrere Threads schreiben / lesen.

## Unbounded Buffer (2)

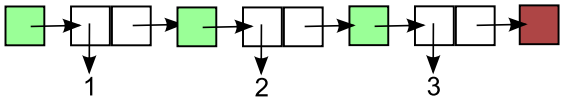
**Veränderbarer Strom**

- ähnlich zu Listen, aber:
- jeder Tail eine `MVar` (dadurch veränderbar)
- leerer Strom = leere `MVar`

```
type Strom a =  MVar (SCons a)
data SCons a =  SCons a (Strom a)
```

Strom der 1,2 und 3 enthält:

```
do ende <- newEmptyMVar
   drei <- newMVar (SCons 3 ende)
   zwei <- newMVar (SCons 2 drei)
   eins <- newMVar (SCons 1 zwei)
   return eins
```

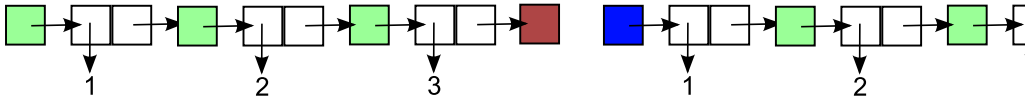





## Auf Strömen operieren

**Strom ausdrucken**

```
type Strom a =  MVar (SCons a)
data SCons a =  SCons a (Strom a)
printStrom strom =
   do
    test <- isEmptyMVar strom
    if test then putStrLn "[]"+
     else do
       SCons el tl <- readMVar strom
       putStr (show el ++ ":")
       printStrom tl
```

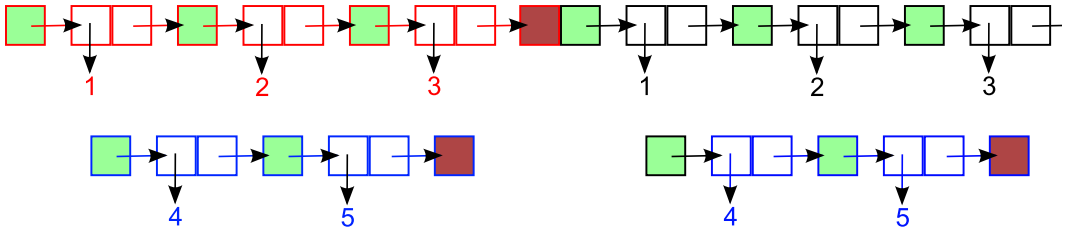





## Ströme aneinanderhängen

```
appendStroeme strom1 strom2 =
 do
  test <- isEmptyMVar strom2
  if test then return strom1
    else do
      kopf2 <- readMVar strom2
      ende <- findeEnde strom1
      putMVar ende kopf2
      return strom1
```

```
findeEnde strom =
 do
  test <- isEmptyMVar strom
  if test then return strom
    else do
     SCons hd tl <- readMVar strom
     findeEnde tl
```

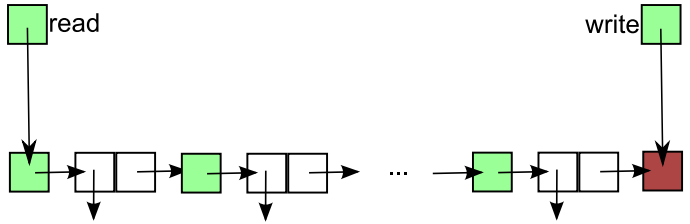





## Kanal

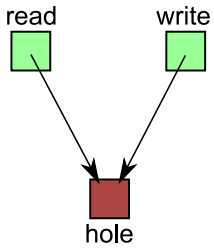
Zwei `MVars`, die auf Lese- und Schreibe-Ende des Stroms zeigen
(Die beiden `MVars` synchronisieren den Zugriff)

```
type Kanal a  = (MVar (Strom a), -- Lese-Ende
                 MVar (Strom a)) -- Schreibe-Ende
```



## Leeren Kanal erzeugen

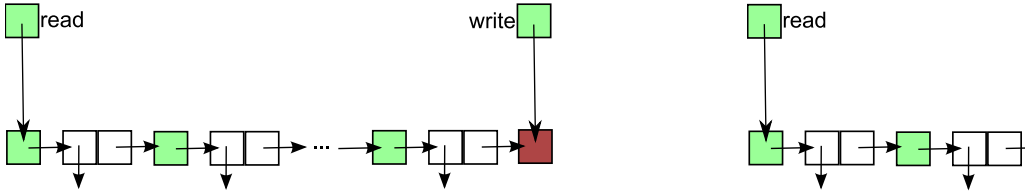
```
neuerKanal :: IO (Kanal a)
neuerKanal =
  do
   hole  <- newEmptyMVar
   read  <- newMVar hole
   write <- newMVar hole
   return  (read, write)
```






## Auf Kanal schreiben

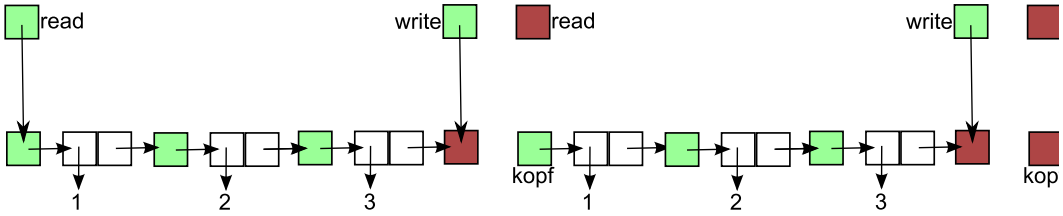
```
schreibe :: Kanal a -> a -> IO ()
schreibe (read,write) val =
 do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar write
  putMVar old_hole (SCons val new_hole)
  putMVar write new_hole
```






## Vom Kanal lesen

```
lese :: Kanal a -> IO a
lese (read,write) =
  do
   kopf <- takeMVar read
   (SCons val stream) <- takeMVar kopf
   putMVar read stream
   return val
```

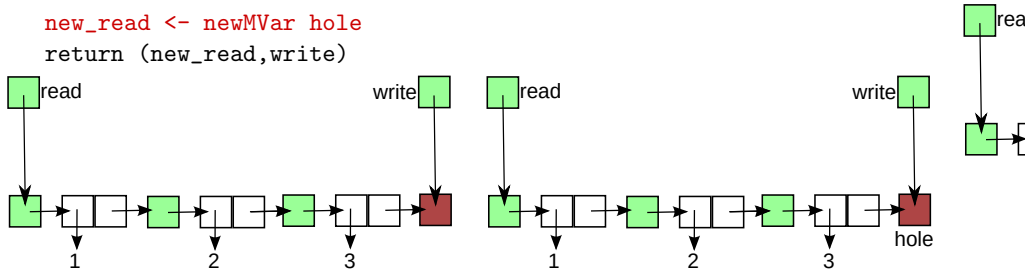





## Generalisierung zu Multicast-Kanälen

Funktion `dupliziere` zum Duplizieren des Kanals, sodass alle schreibe-Zugriffe von nun an auch im neuen Kanal vorhanden sind.

```
dupliziere :: Kanal a -> IO (Kanal a)
dupliziere (read,write) =
 do
   hole <- readMVar write
   new_read <- newMVar hole
   return (new_read,write)
```



## Generalisierung zu Multicast-Kanälen (2)

Funktioniert nur, wenn man die `lese`-Operation anpasst:

```
lese :: Kanal a -> IO a
lese (read,write) =
  do
   kopf <- takeMVar read
   (SCons val stream) <- readMVar kopf -- readMVar statt takeMVar
   putMVar read stream
   return val
```



## Undo einer Lese-Operation?

```
undoLese :: Kanal a -> a -> IO a
undoLese (read,write) val =
  do
   new_read <- newEmptyMVar
   hole <- takeMVar read
   putMVar new_read (SCons val hole)
   putMVar read new_read
```

funktioniert nicht!

Betrachte Fall: Kanal ist leer und eine lese-Operation wartet bereits



## Kanäle in der Bibliothek

- Die gezeigten Kanäle sind in der Bibliothek

  `Control.Concurrent.Chan`

  bereits definiert

- Der Datentyp ist `Chan a`

- Die Operationen heißen `newChan, writeChan, readChan, dupChan`



## Beispiel: Arztpraxis

Eine Gemeinschaftspraxis bestehe aus $n$ Ärzten, die in FIFO-Reihenfolge auf Patienten warten.

Ankommende Patienten reihen sich in FIFO-Ordnung in eine Warteschlange ein.

Die Arzthelferin am Empfang verteilt je einen Patienten am Anfang der „Patientenschlange“ auf einen Arzt am Anfang der „Ärzteschlange“.

Wenn ein Arzt mit seiner Behandlung fertig ist, reiht er sich hinten in die „Ärzteschlange“ ein.

## Beispiel: Arztpraxis (2)

### Hauptthread

```
> arztpraxis n m =
>  do
>   outputsem <- newEmptyMVar
>   patienten_Kanal <- neuerKanal
>   forkIO (mapM_(schreibe patienten_Kanal) [1..m])
>   aerzte_Kanal <- neuerKanal
>   forkIO (mapM_(schreibe aerzte_Kanal) [1..n])
>   ordne patienten_Kanal aerzte_Kanal outputsem
```



## Beispiel: Arztpraxis (3)

### Arzthelferin (ordne)

```
> ordne patienten aerzte outputsem =
>  do
>   patient <- lese patienten
>   arzt <- lese aerzte
>   forkIO (behandle arzt patient aerzte outputsem)
>   ordne patienten aerzte outputsem
```



## Beispiel: Arztpraxis (2)

### Behandlung

```
> behandle arzt patient aerzte outputsem =
>  do
>   j <- randomIO
>   let i = (abs j `mod` 10000000) `div` 1000000
>   atomar outputsem (
>     putStrLn
>      ("Arzt " ++ show arzt ++ " behandelt Patient "
>        ++ show patient ++ " in " ++ show i ++ " Sekunden")
>     )
>   threadDelay (i*1000000)
>   schreibe aerzte arzt
```



## Beispiel: Arztpraxis (3)

Problem bei bisherigen Lösung:

- Die Funktion `ordne` terminiert nicht, wenn alles verteilt wurde.
- Führt zum Deadlock
- Wünschenswert: Operation um Kanal zu schließen.
- Abhilfe: Markiere das Ende

## Beispiel: Arztpraxis (4)

```
> arztpraxis n m = do
>   outputsem <- newEmptyMVar
>   patienten_Kanal <- neuerKanal
>   forkIO (mapM_ (schreibe patienten_Kanal)
>                 ([Just i | i <- [1..m]] ++ [Nothing]))
>   aerzte_Kanal <- neuerKanal
>   forkIO (mapM_(schreibe aerzte_Kanal) [1..n])
>   ordne n patienten_Kanal aerzte_Kanal outputsem

> ordne n patienten aerzte outputsem = do
>   p <- lese patienten
>   case p of
>    Nothing -> sequence_(replicate n (lese aerzte))
>    Just p -> do
>                arzt <- lese aerzte
>                forkIO (behandle arzt patient aerzte outputsem)
>                ordne n patienten aerzte outputsem
```



## Nichtdeterministisches Mischen

- Gegeben: Zwei Listen $xs$ und $ys$
- Mische beide Listen zu einer Liste zusammen, so dass
  - die relative Sortierung von $xs$ und $ys$ erhalten bleibt.
  - Wenn eine Liste partiell ist (d.h. ein Tail ist $\bot$)
    dann erscheinen die Elemente der anderen Liste

Sowohl `ndMerge (1:2:bot) [3..]` als auch `ndMerge [3..] (1:2:bot)`

liefern eine unendlich lange Liste

,

Sequentiell: Nicht möglich! (Auswertung bleibt stecken beim $\bot$)



## Nichtdeterministisches Mischen (2)

Mischen mit Kanal:

- Schreibe beide nebenläufig auf den Kanal
- Hauptthread liest den Kanal aus in eine Liste

```
ndMerge xs ys =
  do
   chan <- neuerKanal
   id_s <- forkIO (schreibeListeAufKanal chan xs)
   id_t <- forkIO (schreibeListeAufKanal chan ys)
   leseKanal chan False
```



## Nichtdeterministisches Mischen (3)

```
schreibeListeAufKanal chan []     = schreibe chan (True,undefined)
schreibeListeAufKanal chan (x:xs) =
 do
  schreibe chan (False,x)
  schreibeListeAufKanal chan xs

leseKanal chan flag =
  do
   (flag1,el) <- lese chan
   if flag1 && flag then return [] else
    do
     rest <- unsafeInterleaveIO (leseKanal chan (flag || flag1))
     if flag1 then return rest else return (el:rest)
```

## Paralleles Oder

Sequentielles Oder:

```
a || b = if a then True else b
```

| a | b | a \|\| b |
|---|---|---|
| True | $s$ | True |
| False | $s$ | $s$ |
| $\bot$ | $s$ | $\bot$ |

$s \in \{\bot, \texttt{False}, \texttt{True}\}$

Gewünscht: Paralleles Oder mit

| a | b | por a b |
|---|---|---|
| True | $s$ | True |
| $s$ | True | True |
| False | $s$ | $s$ |
| $\bot$ | $s$ | $\bot$ |

Sequentiell: nicht möglich!



## Paralleles Oder (2)

```
por :: Bool -> Bool -> IO Bool
por s t = do
  ergebnisMVar <- newEmptyMVar
  id_s <- forkIO (if s then (putMVar ergebnisMVar True)
                  else (putMVar ergebnisMVar t)
  id_t <- forkIO (if t then (putMVar ergebnisMVar True)
                  else (putMVar ergebnisMVar s)
  ergebnis <- takeMVar ergebnisMVar
  killThread id_s
  killThread id_t
  return ergebnis
```



## McCarthys amb

- „ambigious choice“
- binärer Operator, der beide Argumente parallel auswertet
- Falls eine der beiden Auswertungen ein Ergebnis liefert, wird dies als Gesamtresultat übernommen.



## McCarthys amb (2)

```
amb :: a -> a -> IO a
amb s t =
 do
  ergebnisMVar <- newEmptyMVar
  id_s <- forkIO (let x = s in seq x (putMVar ergebnisMVar x))
  id_t <- forkIO (let x = t in seq x (putMVar ergebnisMVar x))
  ergebnis <- takeMVar ergebnisMVar
  killThread id_s
  killThread id_t
  return ergebnis
```

## McCarthys amb (3)

Semantisch

$$\texttt{amb}\ s\ t = \begin{cases} t, \text{ wenn } s \text{ nicht terminiert} \\ s, \text{ wenn } t \text{ nicht terminiert} \\ s \text{ oder } t, \text{ wenn } s \text{ und } t \text{ terminieren} \end{cases}$$



## Kodierung anderer Operatoren mit amb

- Paralleles Oder:

```
por2 :: Bool -> Bool -> IO Bool
por2 s t = amb (if s then True else t) (if t then True else s)
```

- Nichtdeterministische Auswahl („erratic choice“):

```
choice :: a -> a -> IO a
choice s t = do res <- (amb (\x -> s) (\x -> t))
                return (res ())
```

- Nichtdeterministische Auswahl („demonic choice“):

```
dchoice :: a -> a -> IO a
dchoice s t = do res <- amb (seq s t) (seq t s)
                 return res
```



## Erweiterung auf $n$ Argumente

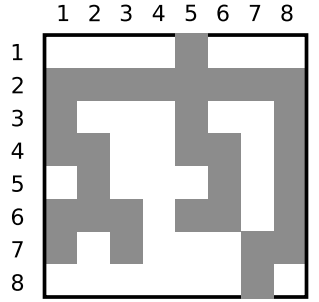
Statt zwei Argumente, eine Liste von Argumenten

```
ambList [x] = return x
ambList (x:xs) =
 do
  l <- unsafeInterleaveIO (ambList xs)
  amb x l
```



## Beispiel: Parallele Suche

Gegeben: Labyrinth
Gesucht: Weg vom Eingang (oben) zum Ausgang (unten)



## Beispiel: Parallele Suche (2)

```
data Suchbaum a = Ziel a | Knoten a [Suchbaum a]

labyrinth =
 let
    kn51 = Knoten (5,1) [kn52]
    kn52 = Knoten (5,2) [kn42,kn53,kn62]
    ...
    kn78 = Ziel (7,8)
 in kn51
```



## Beispiel: Parallele Suche (3)

```
suche =
 do
  result <- ndBFsearch [] labyrinth
  print result

ndBFsearch res (Ziel a)      =
 return (reverse $ a:res)

ndBFsearch res (Knoten a []) =
 do
  yield
  ndBFsearch res (Knoten a [])

ndBFsearch res (Knoten a nf) =
 do
  nf' <- mapM (unsafeInterleaveIO . ndBFsearch (a:res)) nf
  ambList nf'
```



## Futures

- Futures = Variablen deren Wert am Anfang unbekannt ist, aber in der Zukunft verfügbar wird, sobald zugehörige Berechnung beendet ist.
- In Haskell: Jede Variable "eine Future", da verzögert ausgewertet wird.
- Nebenläufige Futures = Wert der Future wird durch nebenläufige Berechnung ermittelt.



## Explizite / Implizite Futures

**Explizite Futures**

- Wert einer Future muss explizit angefordert werden
- Wenn Thread den Wert benötigt, führt er eine `force`-Operation aus: Warte bis Wert berechnet ist.

**Implizite Futures**

- keine `force`-Operation
- Auswertung fordert Wert automatisch an, wenn er benötigt wird.

Programmierung mit impliziten Futures komfortabler!

## Explizite Futures mit MVars

```
type EFuture a = MVar a

efuture :: IO a -> IO (EFuture a)
efuture act =
 do  ack <- newEmptyMVar
     forkIO (act >>= putMVar ack)
     return ack

force :: EFuture a -> IO a
force = readMVar
```



## Beispiel: Parallele Baumsumme

```
data BTree a =
   Leaf a
 | Node a (BTree a) (BTree a)

treeSum (Leaf a)     = return a
treeSum (Node a l r) =
 do
  futl <- efuture (treeSum l)
  futr <- efuture (treeSum r)
  resl <- force futl
  resr <- force futr
  let result = (a + resl + resr)
   in seq result (return result)
```



## Implizite Futures: unsafeInterleaveIO

```
future :: IO a -> IO a
future code = do
 ack <-newEmptyMVar
 thread <- forkIO (code >>= putMVar ack)
 unsafeInterleaveIO
   (do
     result <- takeMVar ack
     killThread thread
     return result)
```



## Baumsumme mit impliziten Futures

```
treeSum (Leaf a)     = return a
treeSum (Node a l r) = do
  futl <- future (treeSum l)
  futr <- future (treeSum r)
  let result = (a + futl + futr)
  in seq result (return result)
```

## Die Async-Bibliothek

Bibliothek `Control.Concurrent.Async`

- ähnlicher Mechanismus wie explizite Futures
- statt Futures: „asynchrone Berechnungen“
- statt `force` nun `wait`
- Verpacken des Ergebnisses durch zusätzlichen Datenkonstruktor



## Implementierung: Basis

```
data Async a = Asynch (MVar a)

async :: IO a -> IO (ASync a)
async action = do
 var <- newEmptyMVar
 forkIO (action >>= \r -> putMVar var r)
 return (Async var)

wait :: Async a -> IO a
wait (Async var) = readMVar var
```



## Beispiel zur Verwendung

```
import Control.Concurrent.Async
import Control.Concurrent
import GetURL
import qualified Data.ByteString  as B
main = do
 a1 <- async (getURL "http://www2.tcs.ifi.lmu.de/~letz/informationen.shtml")
 a2 <- async (getURL "http://www.ifi.lmu.de")
 r1 <- wait a1
 r2 <- wait a2
 print (B.length r1,B.length r2)
```



## Anmerkungen

- Die Async-Bibliothek hat noch mehr Features und Funktionalitäten
- Insbesondere: Behandlung und Weitergabe von Exceptions
- Für Details: siehe Buch von Simon Marlow
- Die echte Implementierung verwendet keine MVars, sondern STM-Haskell