

3.5 Nebenläufige Programmierung in Haskell – Concurrent Haskell

In diesem Abschnitt behandeln wir eine Erweiterung von Haskell um nebenläufige Konstrukte. Vorher machen wir jedoch einen kurzen Einschub, der auch das sequentielle IO von Haskell betrifft.

3.5.1 Einschub: Verzögern innerhalb der IO-Monade

Wir betrachten ein Problem beim monadischen Programmieren. Wir schauen uns die Implementierung des `readFile` an, welches den Inhalt einer Datei ausliest. Hierfür werden intern `Handles` benutzt. Diese sind im Grunde „intelligente“ Zeiger auf Dateien. Für `readFile` wird zunächst ein solcher Handle erzeugt (mit `openFile`), anschließend der Inhalt gelesen (mit `leseHandleAus`).

```
-- openFile :: FilePath -> IOMode -> IO Handle
-- hGetChar :: Handle -> IO Char

readFile :: FilePath -> IO String
readFile path =
  do
    handle <- openFile path ReadMode
    inhalt <- leseHandleAus handle
    return inhalt
```

Es fehlt noch die Implementierung von `leseHandleAus`. Diese Funktion soll alle Zeichen vom `Handle` lesen und anschließend diese als Liste zurückgeben und den Handel noch schließen (mit `hClose`). Wir benutzen außerdem die vordefinierten Funktion `hIsEOF :: Handle -> IO Bool`, die testet ob das Dateieende erreicht ist und `hGetChar`, die ein Zeichen vom Handle liest.

Ein erster Versuch führt zur Implementierung:

```
leseHandleAus handle =
  do
    ende <- hIsEOF handle
    if ende then hClose handle >> return []
    else
      do
        c <- hGetChar handle
        cs <- leseHandleAus handle
        return (c:cs)
```

Diese Implementierung funktioniert, ist allerdings sehr speicherlastig, da das letzte `return` erst ausgeführt wird, nachdem auch der rekursive Aufruf durchgeführt wurde. D.h. wir lesen die gesamte Datei aus, bevor wir irgendwas zurückgeben. Dies ist unabhängig davon, ob wir eigentlich nur das erste Zeichen der Datei oder alle Zeichen benutzen wollen. Für eine verzögert auswertende Programmiersprache und zum eleganten Programmieren ist dieses Verhalten nicht gewünscht. Deswegen benutzen wir die Funktion `unsafeInterleaveIO :: IO a -> IO a`, die die strenge Sequentialisierung der IO-Monade aufbricht.

```
unsafeInterleaveIO a = return (unsafePerformIO a)
```

Die Implementierung von `leseHandleAus` ändern wir nun ab in:

```
leseHandleAus handle =
do
  ende <- hIsEOF handle
  if ende then hClose handle >> return []
  else
    do
      c <- hGetChar handle
      cs <- unsafeInterleaveIO (leseHandleAus handle)
      return (c:cs)
```

Nun liefert `readFile` schon Zeichen, bevor der komplette Inhalt der Datei gelesen ist. Beim Testen im Interpreter sieht man den Unterschied.

Mit der Version ohne `unsafeInterleaveIO`:

```
*Main> writeFile "LargeFile" (concat [show i | i <- [1..100000]])
*Main> readFile "LargeFile" >>= print . head
'1'
(7.09 secs, 263542820 bytes)
```

Mit Benutzung von `unsafeInterleaveIO`

```
*Main> writeFile "LargeFile" (concat [show i | i <- [1..100000]])
*Main> readFile "LargeFile" >>= print . head
'1'
(0.00 secs, 0 bytes)
```

3.5.2 Concurrent Haskell

Concurrent Haskell ist eine Erweiterung von Haskell um Konstrukte zur nebenläufigen Programmierung. Diese Erweiterung wurde als notwendig empfunden, um IO-lastige Real-World-Anwendungen, wie. z.B. Graphische Benutzeroberflächen oder diverse Serverdienste (z.B. http-Server) in Haskell zu implementieren.

Die neuen Konstrukte sind

- Nebenläufige Threads und Konstrukte zum Erzeugen solcher Threads
- Konstrukte zur Kommunikation zwischen Threads und zur Synchronisation von Threads.

Insgesamt wurden dafür folgende neue primitive Operationen zu Haskell hinzugefügt, die innerhalb der IO-Monade verwendet werden dürfen.

Zur Erzeugung von nebenläufigen Threads existiert die Funktion

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

Diese Funktion erwartet einen Ausdruck vom Typ `IO ()` und führt diesen in einem nebenläufigen Thread aus. Das Ergebnis ist eine eindeutige Identifikationsnummer für den erzeugten Thread. D.h. aus Sicht des Hauptthreads liefert `forkIO s` sofort ein Ergebnis zurück. Im GHC ist zusätzlich eine Funktion `killThread :: ThreadId -> IO ()` implementiert, die es erlaubt einen nebenläufigen Thread anhand seiner `ThreadId` zu beenden. Ansonsten führt das Beenden des Hauptthreads auch immer zum Beenden aller nebenläufigen Threads.

Im GHC steht zusätzlich noch die sehr ähnliche Funktion

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

zur Verfügung. Der Unterschied zwischen `forkIO` und `forkOS` besteht darin, wer den erzeugten nebenläufigen Thread verwaltet. Während mit `forkIO` erzeugte Threads vom Haskell Runtime-System erzeugt und verwaltet werden, werden mit `forkOS` erzeugte Threads vom Betriebssystem verwaltet.

Zur Synchronisation und Kommunikation zwischen mehreren Threads wurde ein neuer Datentyp `MVar` (mutable variable) eingeführt. Hierbei handelt es sich um Speicherplätze die im Gegensatz zum Datentyp `IORef` auch zur Synchronisation verwendet werden können.

Für den Datentyp stehen drei Basisfunktionen zur Verfügung:

- `newEmptyMVar :: IO (MVar a)` erzeugt eine leere `MVar`, die Werte vom Typ `a` speichern kann.
- `takeMVar :: MVar a -> IO a` liefert den Wert aus einer `MVar` und hinterlässt die Variable leer. Falls die entsprechende `MVar` leer ist, wird der Thread, der die `MVar` lesen möchte, solange blockiert, bis die `MVar` gefüllt ist.

Wenn mehrere Threads ein `takeMVar` auf die gleiche (zunächst leere) Variable durchführen, so wird nachdem die Variable einen Wert hat nur ein Thread bedient. Die anderen Threads warten weiter. Die Reihenfolge hierbei ist first-in-first-out (FIFO), d.h. jener Thread der zuerst das `takeMVar` durchführt, wird zuerst bedient.

- `putMVar :: MVar a -> a -> IO ()` speichert den Wert des zweiten Arguments in der übergebenen Variablen, wenn diese leer ist. Falls die Variable bereits durch einen Wert belegt ist, wartet der Thread solange, bis die Variable leer ist. Genau wie bei `takeMVar` wird bei gleichzeitigem Zugriff von mehreren Threads auf die gleiche Variable mittels `putMVar` der Zugriff in FIFO-Reihenfolge abgearbeitet.

Zusätzlich gibt es weitere darauf aufbauende Funktionen, z.B. die Funktion `newMVar :: a -> MVar a`, die eine neue `MVar` erzeugt und den Wert des Arguments in die Variable schreibt.

Eine `MVar` implementiert direkt eine Semaphore: Erstellen einer `MVar` kommt dem Erzeugen der Semaphore gleich, das Belegen der Ressource geschieht durch `putMVar`, das Freigeben mittels `takeMVar`.

3.5.3 Exklusiver Zugriff – Atomare Aktionen

Wir betrachten zunächst das folgende Beispiel

```
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) -- 1. nebenl"aufiger Thread
  forkIO (echo 2) -- 2. nebenl"aufiger Thread
  block    -- Blockiert den Hauptthread

block = do block
```

Bei Aufruf von `zweiEchos` werden zwei Threads gestartet, die jeweils ein Echo der Tastatureingaben (zeilenweise) durchführen. Da keinerlei Synchronisation zwischen beiden Prozessen stattfindet, geraten die Ein- und Ausgaben wild durcheinander:

```
*Main> zweiEchos
EinEgianbgea bfeu efru eTrh rTeharde1a:d2:Thread1?
Letzte Eingabe fuer Thread1:Thread1?
Eingabe fuer Thread1:Thread1 fragt?
Letzte Eingabe fuer Thread2:Thread1 fragt?
Eingabe fuer Thread2:Jetzt Thread 2?
Letzte Eingabe fuer Thread1:Jetzt Thread 2?
```

Die Frage nach der Eingabe und das eigentliche Einlesen des Textes von der Standardeingabe ist ein *kritischer Abschnitt*, solche Programmabschnitte sollten zur gleichen Zeit nur von einem Thread betreten werden.

Mithilfe von `MVars` kann der Zugriff auf eine Ressource geschützt werden, so dass alleinig ein Thread zur gleichen Zeit zugreifen kann, vorausgesetzt alle Zugriffe entsprechen der zu definierenden Konvention. Für das Echo-Beispiel definieren wir eine eigene Funktion `atomar`, welche die gegebenen Aktionen nur ausführt während sie „im Besitz“ der `MVar` ist:

```
atomar mvar aktionen =
do
  putMVar mvar () -- MVar belegen
  result <- aktionen -- gesch"utzte Aktionen
  takeMVar mvar -- MVar entleeren
  return result

echoS sem i =
do
  result <- atomar sem (
    do
      putStr $ "Eingabe fuer Thread" ++ show i ++ ":"
      line <- getLine
      return line
    )
  atomar sem
  (putStrLn $ "Letzte Eingabe fuer Thread" ++ show i ++ ":" ++ result)
echoS sem i

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

Bei Aufruf von `zweiEchosS` laufen die Aktionen jeweils `atomar` ab und es entsteht kein Durcheinander:

```
*Main> zweiEchosS
Eingabe fuer Thread1:Eingabe fuer Thread 1
Eingabe fuer Thread2:Eingabe fuer Thread 2
Letzte Eingabe fuer Thread1:Eingabe fuer Thread 1
Letzte Eingabe fuer Thread2:Eingabe fuer Thread 2
Eingabe fuer Thread1:
```

3.5.4 Speisende Philosophen

Im folgenden zeigen wir eine Implementierung eines Standardbeispiels der nebenläufigen Programmierung. Gegeben sei ein runder Tisch, an dem n speisende Philosophen im Kreis sitzen. Zwischen den Philosophen liegt jeweils eine Gabel. Ein Philosoph kann entweder essen oder denken, allerdings kann er nur essen wenn er zwei Gabeln (die linke und die rechte) besitzt.

Eine Gabel implementieren wir durch eine `MVar ()`. Die Funktion `philosoph`, die dem i -ten Philosophen entspricht, und das Erzeugen der Philosophen und Gabeln mittels `philosophen` kann dann wie folgt programmiert werden:

```
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

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, dass in dieser Variante ein Deadlock auftreten kann, falls alle Philosophen die linke Gabel in der Hand haben.

Eine einfache Deadlock-freie Implementierung kann erzielt werden, indem immer nur ein Philosoph zur gleichen Zeit an alle Gabeln darf. Dies wird durch eine `MVar` als Semaphore realisiert.

```
philosoph' sem i gabeln =
  do
    let n = length gabeln
        atomar sem (do
            takeMVar $ gabeln!!i
            putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel ...\n"
```

```

        takeMVar $ gabeln!!(mod (i+1) n)
        putStr $ "Philosoph " ++ show i ++ " isst ...\n"
        putMVar (gabeln!!i) ()
        putMVar (gabeln!!(mod (i+1) n)) ()
    )
    putStr $ "Philosoph " ++ show i ++ " denkt ...\n"
    philosoph' sem i gabeln

philosophen' n =
  do
    sem <- newEmptyMVar
    gabeln <- sequence $ replicate n (newMVar ())
    sequence_ [forkIO (philosoph' sem i gabeln) | i <- [0..n-1]]
  block

```

Diese Variante ist zwar Deadlock-frei, allerdings nicht fair, denn es kann durchaus passieren, dass ein Philosoph nie essen darf.

3.5.5 Kommunikation zwischen Threads

In diesem Abschnitt wollen wir zeigen, wie Threads Werte miteinander austauschen können

Puffervariable

Eine einfache Kommunikation zwischen Threads führt über eine Puffervariable, hierbei ist ein Thread der „Erzeuger“, d.h. er erzeugt (berechnet) Werte und schreibt sie in einen Speicherplatz, während der zweite Thread (der „Verbraucher“) die Werte aus der Variablen liest und sie verwendet.

Um zu verhindern, dass der Erzeuger den Speicherplatz überschreibt, bevor der Verbraucher ihn gelesen hat, wird eine `MVar` benutzt, um so die Kommunikation zwischen beiden Threads zu synchronisieren. Somit lässt sich ein solcher Puffer direkt durch eine `MVar` implementieren.

```
type Puffer a = MVar a
```

Ein neuer leerer Puffer wird durch Erzeugen einer leeren `MVar` implementiert:

```
newPuffer :: IO (Puffer a)
newPuffer = newEmptyMVar
```

Das Schreiben in den Puffer entspricht der Funktion `putMVar`, wobei zu beachten ist, dass solange der Puffer besetzt ist, d.h. der Verbraucher nicht gelesen hat, der Erzeuger blockiert wird, d.h. wartet.

```
writeToPuffer :: Puffer a -> a -> IO ()
writeToPuffer = putMVar
```

Beim Lesen aus dem Puffer verwendet der Verbraucher die Funktion `takeMVar`, d.h. solange der Erzeuger noch keinen Wert liefert, wartet der Verbraucher.

```
readFromPuffer :: Puffer a -> IO a
readFromPuffer = takeMVar
```

Kanäle beliebiger Länge

Über den Puffer kann immer nur ein Wert ausgetauscht werden, das ist i.A. nicht sinnvoll, wenn z.B. der Erzeuger wesentlich schneller ist als der Verbraucher, oder wenn es mehrere Erzeuger und Verbraucher gibt. Hierfür eignen sich Kanäle. Ein (FIFO)-Kanal besteht aus einer Liste von Elementen, wobei an einem Ende Werte angehängt, am anderen Ende Elemente gelesen (und entfernt) werden können.

Als Schnittstelle sollte somit verfügbar sein:

- ein Typ für den Kanal der polymorph über dem Elementtyp ist:
`type Kanal a`
- eine Funktion zum Erzeugen eines neuen, leeren Kanals:
`neuerKanal :: IO (Kanal a)`
- eine Funktion zum Anhängen eines Elements an den Kanal:
`schreibe :: Kanal a -> a -> IO ()`
- eine Funktion zum Lesen (und Entnehmen) des zuerst eingefügten Elements: `lese :: Kanal a -> IO a`

Zudem sollen keine Fehler auftreten, wenn mehrere Threads vom Kanal lesen, oder in den Kanal schreiben.

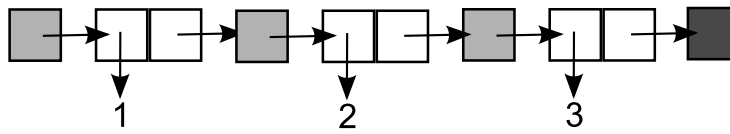
Der eigentliche Strom wird durch eine veränderliche Liste implementiert, indem jeder Tail durch eine `MVar` verkettet wird (d.h. jeder Tail ist veränderbar). Das Ende der Liste („Nil“) ist dann genau eine leere `MVar`:

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

Z.B. kann der Strom der 1,2 und 3 speichert erzeugt werden mit

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

Als Box-and-Pointer-Diagramm, wobei `MVars` graue, leere `MVars` dunkelgraue Kästchen sind, kann dieser Strom wie folgt dargestellt werden:



Ströme sind nun veränderbare Listen. Wir können z.B. zwei Ströme aneinander hängen oder einen Strom ausdrucken:

```
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

listToStrom [] = newEmptyMVar
listToStrom (x:xs) = do
  tl <- listToStrom xs
  v <- newMVar (SCons x tl)
  return v

printStrom strom =
  do
    test <- isEmptyMVar strom
```

```

if test then putStrLn "[]\n"
else do
  SCons el tl <- readMVar strom
  putStr (show el ++ ":")
  printStrom tl

```

Beim Aneinanderhängen werden wirklich die benutzten Speicherplätze verändert, wie der folgende Test im Interpreter zeigt

```

*Main> f <- listToStrom [1..10]
*Main> g <- listToStrom [11..20]
*Main> printStrom f
1:2:3:4:5:6:7:8:9:10: []
*Main> printStrom g
11:12:13:14:15:16:17:18:19:20: []
*Main> h <- appendStroeme f g
*Main> printStrom h
1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20: []
*Main> printStrom f
1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20: []
*Main> printStrom g
11:12:13:14:15:16:17:18:19:20: []

```

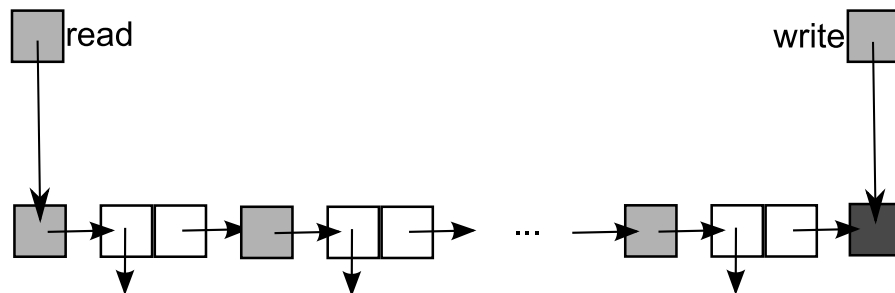
Zur Definition des Kanals werden nun zwei MVars benutzt, die auf das Lese-Ende (links) und auf das Schreibe-Ende (rechts) eines Stroms zeigen. Die Verwendung von MVars sorgt dafür, dass immer nur ein Thread auf ein Ende zugreifen kann:

```

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

```

In Box-and-Pointer-Darstellung sieht ein Kanal somit wie folgt aus:



Eine leerer Kanal wird nun erzeugt, indem 3 MVars erzeugt werden: Eine für das Lese-Ende, eine für das Schreibe-Ende und eine für den leeren Strom:

```

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

```

Schreiben an das Schreibe-Ende und Lesen vom Kanal kann nun wie folgt implementiert werden:

Beim Schreiben wird zunächst eine neue `MVar` erzeugt, die das neue Stromende darstellen soll, anschließend wird das alte Stromende berechnet. Ab diesem Zeitpunkt ist die `MVar write` leer, d.h. andere schreibende Threads müssen warten. Im nächsten Schritt wird erhält die `write` als neuen Wert das neue Stromende (konkurrierende Threads können somit weiterlaufen). Im letzten Schritt wird das ehemalige Listenende durch das neue Element ersetzt.

```

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

```

Beim Lesen wird im ersten Schritt die `MVar read` gelesen, sie ist somit leer andere lesende Threads müssen warten. Im zweiten Schritt wird das erste Stromelement gelesen, anschließend die `MVar read` auf das ehemals zweite Element des Stroms gesetzt. Erst ab diesem Zeitpunkt können konkurrierende Threads fortfahren. Im letzten Schritt wird der Wert zurück gegeben.

```

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

```

Nichtdeterministisches Mischen

Mithilfe des Kanals können wir zwei Listen nichtdeterministisch mischen, indem wir zunächst einen Kanal erzeugen, anschließend für beide Listen einen

nebenläufigen Thread erzeugen, der jeweils die Elemente der Liste auf den Kanal schreibt (mittels `schreibeListeAufKanal`), und im Hauptthread die Elemente vom Kanal lesen und in eine Liste einfügen (mittels `leseKanal`). Um zu erkennen, wann alle Elemente in den Kanal geschrieben wurden, ist die tatsächliche Implementierung etwas komplizierter: Wir schreiben nicht nur die Elemente in den Kanal, sondern für jedes Listenelement e das Paar `(False,e)` in den Kanal. Sobald das letzte Element geschrieben wurde (das Ende der Liste ist erreicht), schreiben wir `(True,bot)` auf den Kanal. Der Leser weiß also, dass keine weiteren Elemente mehr erscheinen, sobald er zweimal `(True,_)` gelesen hat. Zum Bewerkstelligen des verzögerten Auslesens des Kanals benutzen wir `unsafeInterleaveIO`.

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

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)

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

Eine solche Mischfunktion ist z.B. sehr nützlich, wenn Ereignisse von verschiedenen Erzeugern geliefert werden (z.B. Tastaturereignisse, Mausereignisse, usw.) und diese nacheinander vom Verbraucher (z.B. dem Betriebssystem) verarbeitet werden sollen.

Eine weitere Eigenschaft dieser Mischfunktion ist, dass sie zum einen für unendliche Listen definiert und zum anderen auch dann noch etwas liefert, wenn eine der beiden Eingabelisten nicht definiert ist, wie die folgenden Test im Interpreter zeigen:

```
*Main> ndMerge (1:2:(let bot = bot in bot) ) [3..] >>= print
```

```
[1,3,2,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25...
*Main> ndMerge [3..] (1:2:(let bot = bot in bot) ) >>= print
[3,1,4,2,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25
```

Aufbauend auf Kanälen können weitere Standardkonstrukte der nebenläufigen Programmierung implementiert werden, die wir jedoch nicht weiter betrachten werden.

3.5.6 Kodierung nichtdeterministischer Operationen

Mithilfe von Concurrent Haskell ist es möglich nichtdeterministische Operationen innerhalb der IO-Monade zu definieren.

Paralleles Oder

Während in einer sequentiellen Programmiersprache die Implementierung des parallelen Oders nicht möglich ist, kann dies mit den nebenläufigen Konstrukten von Concurrent Haskell programmiert werden.

Wir betrachten zunächst das Verhalten des (sequentiellen) Oders in Haskell, wobei s ein Ausdruck aus $\{\text{True}, \text{False}, \perp\}$ sei.

a	b	a b
False	s	s
True	s	True
\perp	s	\perp

Ein *paralleles Oder* stellt die Anforderung das $a \vee b$ zu **True** auswertet, sofern a oder b zu **True** auswerten. Diese ist in der letzten Zeile verletzt, da `bot || True` nicht terminiert.

Die Implementierung des parallelen Oder mittels Concurrent Haskell basiert auf der folgenden Idee: Es werden beide Argumente des Oder nebenläufig ausgewertet und sobald eines der beiden zu **True** ausgewertet ist, wird dieser Wert als Resultat des parallelen Oders übernommen. Falls eine Argumentauswertung mit **False** endet ist das Ergebnis gerade das andere Argument. Zur Synchronisation der beiden nebenläufigen Auswertungen benutzen wir eine **MVar**. Diese wird zunächst leer erzeugt, anschließend werden die beiden nebenläufigen Auswertungen gestartet, die ihr Ergebnis in die **MVar** schreiben. Der Hauptthread liest nun die **MVar** (d.h. er wartet solange, bis einer der beiden Threads etwas in die **MVar** geschrieben hat) und liefert das erhaltene Resultat als Ergebnis zurück. Vorher beendet er noch beide Threads, damit keine unnötigen Threads mehr laufen. Insgesamt ergibt das den Code:

```

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

```

Das Verhalten von `por` lässt sich im Interpreter testen:

```

*Main> por False False >>= print
False
*Main> por False True >>= print
True
*Main> por True False >>= print
True
*Main> por True True >>= print
True
*Main> por True (let bot = bot in bot) >>= print
True
*Main> por (let bot = bot in bot) True >>= print

```

Die Implementierung hat somit das gewünschte Verhalten. Man beachte, dass der Wert von `por s t` nur vom Wert der Argumente `s` und `t` abhängt, d.h. eigentlich ist das parallele Oder kein echter nichtdeterministischer Operator. Solche Operatoren werden auch als *schwach nichtdeterministisch* bezeichnet. Es gibt keinen echten Grund das parallele Oder innerhalb der IO-Monade zu definieren, d.h. die referentielle Transparenz ist durch die Definition

```

safePor :: Bool -> Bool -> Bool
safePor s t = unsafePerformIO $ por s t

```

nicht verletzt. Allerdings geben die GHC-Entwickler keine Garantie, dass `unsafePerformIO` zusammen mit Concurrent Haskell wirklich wie erwartet funktioniert.

McCarthy's amb

Wir betrachten nun einen (*stark*) nichtdeterministischen Operator. Der vom Lisp-Erfinder John McCarthy vorgeschlagene Operator `amb` (abgeleitet von „ambiguous choice“) erwartet zwei Argumente, wertet diese parallel (oder nebenläufig) aus. Wenn eine der beiden Auswertungen mit einem Wert endet, wird dieser als Gesamtergebnis übernommen.

Die Implementierung mittels Concurrent Haskell als Operator in der IO-Monade ist ähnlich zur Implementierung des parallelen Oders:

```
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
```

Man beachte, dass `seq` die Auswertung zur WHNF erzwingt. Dies ist notwendig, da ansonsten der unausgewertete Ausdruck in die `MVar` geschrieben würde.

Das Ergebnis eines Aufrufs `amb s t` hängt nicht alleinig von den Werten der Argumente `s` und `t` ab, denn falls beide Argumente zu einem „echten“ Wert auswerten können, kann sowohl der Wert von `s` oder aber auch der Wert von `t` als Gesamtergebnis gewählt werden. Die Entscheidung hängt dann lediglich davon ab, welcher der beiden nebenläufigen Threads schneller fertig ist, was wiederum vom Scheduling und der damit verbundenen Ressourcenverteilung abhängig ist. Semantisch lässt sich das Ergebnis der Auswertung des `amb`-Konstruktes beschreiben als:

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

Der Operator `amb` ist aus verschiedenen Sichtweisen interessant. Er ermöglicht die Kodierung verschiedener nichtdeterministischer Operatoren, z.B. kann das parallele Oder mithilfe von `amb` definiert werden:

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

Außerdem kann ein `choice`, welches willkürlich zwischen seinen beiden Argumenten wählt mittels `amb` definiert werden:

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

Hierbei zeigt sich jedoch beim Testen im GHCi, dass meist das erste Argument gewählt wird, da anscheinend der zuerst erzeugte Thread auch zuerst Ressourcen erhält.

Wir können den `amb`-Operator auch auf eine Liste von beliebig vielen Argumenten erweitern:

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

Diese Implementierung funktioniert jedoch noch nur auf endlichen Listen, da durch die Verwendung der IO-Monade das Erzeugen der nebenläufigen Threads sequenzialisiert wird. Für eine Liste $a : as$, wird der Ausdruck `amb a l` erst erzeugt, nachdem die durch l zu erzeugenden Threads, wirklich erzeugt wurden.

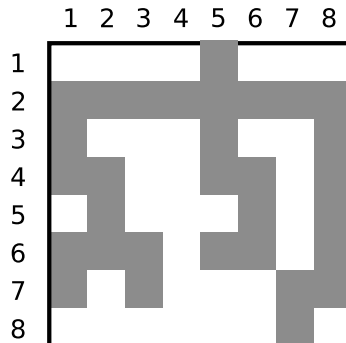
Mithilfe von `unsafeInterleaveIO :: IO a -> IO a` können wir dies (genau wie bei `readFile`) verhindern:

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

Nun wird das `amb` sofort erzeugt, da l sofort einen Wert liefert. Diese Implementierung funktioniert nun auch für unendliche Listen. Z.B. liefert der folgende Aufruf ein Ergebnis im Interpreter:

```
ambList ([let bot = bot in bot] ++ [2..]) >>= print
2
```

Abschließend zum `amb` betrachten wir noch als Beispiel eine „parallelisierte“ Breitensuche, die sich mithilfe des `amb` sehr deklarativ programmieren lässt. Gegeben sei das folgende Labyrinth:



Das Ziel ist vom Eingang (oben) einen Weg zum Ausgang (unten) zu finden. Wir modellieren das Labyrinth mithilfe eines (n-ären) Suchbaums, wobei die Markierung eines Knotens genau den Koordinaten eines Feldes entspricht:

```
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
```

Für die eigentlich Breitensuche implementieren wir die Funktion `ndBFsearch`, die als erstes Argument die Liste der Pfadmarkierungen bis zum aktuellen Knoten und als zweites Argument den aktuellen Knoten erhält.

```
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'
```

Zur Erläuterung von `ndBFsearch`: Wenn die Suche am Ziel ist, dann werden die Pfadmarkierungen als Liste zurück gegeben. Wenn wir an einem Knoten sind, der keine Nachfolger hat, so blockieren wir diese Suche indem wir in eine Endlosschleife gehen (das zusätzliche `yield` ist eine kleine Optimierung: Es erzwingt einen Kontextwechsel, d.h. ein anderer Thread ist erstmal dran). An einem Nichtzielknoten, der noch Nachfolger hat rufen wir rekursiv die Breitensuche für alle Nachfolger auf und verknüpfen die Suchergebnisse mittels `ambList`, d.h. die zuerst erfolgreiche Suche wird als Ergebnis gewählt. Wiederum benutzen wir `unsafeInterleaveIO`, um die rekursiven Such-Aktionen zu erzeugen, aber noch nicht auszuwerten.

Der Aufruf im Interpreter ergibt den gesuchten Weg:

```
*Main> suche  
[(5,1), (5,2), (6,2), (7,2), (8,2), (8,3), (8,4), (8,5), (8,6), (8,7), (7,7), (7,8)]
```