Nebenläufigkeit in Haskell

Sven Eric Panitz

8. Mai 2021

Inhaltsverzeichnis

1 Nebenläufigkeit

Software ist in den meisten Fällen des echten Lebens kein Programm, dass eine Funktionalität bereitstellt, die sich in einer Funktion äußert, die eine Eingabe erhält und ein Ergebnis berechnet. Vielmehr besteht die meiste Software aus vielen nebenläufig ausgeführten Funktionalitäten. Man denke dabei an Dienste, die auf einem Server angeboten werden, wie ein Webserver oder beliebige Webdienste. Diese warten auf Anfragen, die abgearbeitet werden. Im besten Falle geht dieses nebenläufig, so dass während der Bearbeitung einer Anfrage, weitere Anfragen angenommen und abgearbeitet werden können. Auch jedes Programm mit einer grafischen Anwenderoberfläche braucht oft eine Form der Nebenläufigleit, damit die Oberfläche nicht einfriert, wenn komplexe Berechnungen gestartet wurden oder nebenläufig Inhalte über ein Netzwerk geladen werden.

Für die Umsetzung einer Nebenläufigkeit gibt es in den meisten Programmiersprachen die Möglichkeit, Steuerfäden (threads) zu definieren.

Eine Nebenläufigkeit muss notwendiger Weise aber auch wieder die Möglichkeit haben, an bestimmten Punkte nebenläufig gestartete Aktionen zu synchronisieren.

Ebenso wie schon allgemeine I/O-Operationen sich schlecht mit der lazy Auswertungsstrategie vertragen, sind Steuerfäden zunächst etwas, dass schwierig mit Haskell vereinbar ist. Wir werden in diesem Lehrbrief sehen, wie unter Verwendung der Monaden auch eine effektive, einfache und effiziente Umsetzung von Steuerfäden in Haskell funktioniert. Haskell hat seit 1996 die Möglichkeit zur Programmierung nebenläufiger Steuerfäden [JGF96].

Haskell: Concurrent

```
> module Concurrent where
```

Die wichtigste Funktionalität zur Nebenläufigkeit befinden sich in dem Modul: `Control.Concurrent`.

Da die Nebenläufigkeit über Monaden in Haskell integriert ist, importieren wir auch das entsprechende Modul und viele weitere Module, die wir verwenden werden.

Haskell: Concurrent

```
> import Prelude hiding (putStr)

> import Control.Concurrent
> import Control.Concurrent.MVar
> import Control.Parallel
> import Control.Monad
> import Data.List
> import Data.String
> import Control.Exception.Base

> import Data.ByteString (putStr)

> import Network.Socket
> import System.IO  hiding (putStr)
```

1.1 Nichtterminierende Aktionen

Nebenläufigkeit wird besonders dann wichtig, wenn es eine Funktion gibt, die während des ganzen Programmdurchlaufs kontinuierlich durchgeführt wird. Zum Beispiel wenn immer wieder das Wort ›young‹ auf der Kommandozeile ausgegeben wird. Dieses lässt sich zum Beispiuel rekursiv von Hand lösen[1]:

[1] Statt mit `putStr` Strings auf der Kommandozeile auszugeben, verwenden wir das `putStr` für den Typ `ByteString` und wandeln die Strings mit `fromString` in `ByteString` um. Grund ist, dass dieses atomar geschieht und so verschieene `putStr` in unterschiedlichen Steuerfäden sich nicht vermischen.

Haskell: Concurrent

```
> t1 = do
>   putStr$fromString "young"
>   t1
```

Oder man nimmt die Funktion `forever` aus dem Monadenmodul, so dass man ›forever young‹ definiert als:

Haskell: Concurrent

```
> t2  = forever$putStr$fromString "young"
```

Dieses ist ein bisschen so, wie eine `while(true)`-Schleife in imperativen Sprachen.

Dass diese Funktion nicht endet, merkt man spätestens daran, wenn man anschließend noch eine weitere Aktion ausführen möchte.

Haskell: Concurrent

```
> t3 = do
>   forever$print "young"
>   putStr$fromString "fertig"
```

Es wir niemals zur Ausgabe von ›fertig‹ kommen.

Wem die Funktionen `t1`, `t2` und `t3` zu viele Ausgaben macht, der kann eine kleine Verzögerung einbauen. Hierzu dient die Funktion `threadDelay`, die eine Art *sleep* realisiert. Das Argument ist die Verzögerung in Microsekunden.

Haskell: Concurrent

```
> t4 = do
>   forever$ do
>     putStr$fromString "young"
>     threadDelay 1000000
>   putStr$fromString "fertig"
```

So wird also alle Sekunde auf der Kommandozeile das Wort ›young‹ ausgedruckt, aber nie das Wort ›fertig‹.

1.2 Nebenläufigkeit starten

Haskell stellt nun eine einfache Funktion zur Verfügung, um eine Nebenläufigkeit zu starten:

Haskell: Concurrent

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

Es lässt sich also eine IO-Aktion in einem eigenen Steuerfaden starten. Dieser läuft dann nebenläufig, während der eigentliche Steuerfaden ungehindert weiter ausgeführt wird. In unserem Beispiel können wir jetzt die Funktion, die ›forever young‹ sagt, mit `forkIO` in einem eigenen Steuerfaden ausführen.

Haskell: Concurrent

```
> t5  = do
>   forkIO$forever$ do
>     putStr$fromString "young"
>     threadDelay 1000000
>   putStr$fromString "fertig"
```

Starten wir diese Funktion, sehen wir, wie hier tatsächlich die beiden Steuerfäden nebenläufig gehen:

Shell

```
[1 of 1] Compiling Concurrent       ( solution/Concurrent.lhs, interpreted )
Ok, one module loaded.
*Concurrent> t5
youngfertig*Concurrent> youngyoungyoungyoungyoungyoung
```

Soweit so gut. Wir können jetzt Nebenläufigkeit starten, zum Beispiel zwei Steuerfäden, die in unterschiedlichen Zeitabständen unterschiedliche Ausgaben auf der Kommandozeile machen:

Haskell: Concurrent

```
> t6  = do
>   forkIO$forever$ do
>     putStr$fromString "young"
>     threadDelay 1000000
>   forkIO$forever$ do
>     putStr$fromString "forever"
>     threadDelay 1500000
>   putStr$fromString "los gehts!"
```

Ein Aufruf dieser Funktion führt zu folgender Ausgabe:

Shell

```
[1 of 1] Compiling Concurrent       ( solution/Concurrent.lhs, interpreted )
Ok, one module loaded.
*Concurrent> t6
los gehts!youngforever*Concurrent> youngforeveryoungforeveryoungyoungforeveryoung
```

1.3 Synchronisation von Steuerfäden

Mit der Möglichkeit, Steuerfäden zu starten, haben wir nur die halbe Miete für eine nebenläufige Programmierung. Wir brauchen eine Möglichkeit, um Steuerfäden wieder miteinander zu synchronisieren. Viele Steuerfäden laufen ja nicht unendlich, sondern liefern nach einer endlichen Zeit ein Ergebnis einer komplexen Berechnung, oder einer langwierigen Ladefunktion über das Netzwerk. Andere Steuerfäden warten auf diese Ergebnisse, um mit ihnen weiter zu arbeiten.

Das Haskellmodul `Control.Concurrent` bietet vier Untermodule an, um zur Kommunikation und Synchronisation zwischen Steuerfäden eingesetzt zu werden.

- `Control.Concurrent.MVar` für veränderbare Variablen, die über Steuerfädengrenzen hinweg gelesen und geschrieben werden können.
- `Control.Concurrent.Chan` für Kanäle, die auch über veränderbare Variablen realisiert sind.
- `Control.Concurrent.QSem` und `Control.Concurrent.QSemN` für Semaphore, die wir hier nicht weiter betrachten werden.

1.3.1 Veränderbare globale Variablen

Steuerfäden können sich Speicherplätze teilen, in diese Werte legen und wieder herausnehmen. Hierzu dient der polymorphe Typ `(MVar t)`. Er repräsentiert einen Speicherplatz, in dem Daten vom Typ `t` abgelegt werden können. Zum Erzeugen eines solchen Speicherplatzes dienen zwei Funktionen:

Haskell: Concurrent

```
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
```

Die erste der Funktionen dient der Erzeugung eines Speicherplatzes, ohne dass dort direkt Daten hinterlegt werden, die zweite legt direkt in den Speicherplatz Daten ab.

Wichtig ist also, sich klar zu machen, dass diese MVar-Speichervariablen leer sein können, dass dort eventuell noch kein Wert hinterlegt ist.

Wenn wir einen Speicherplatz für eine Zahl benötigen, lässt sich hierfür ein Speicherplatz schaffen und in diesen direkt die 0 ablegen:

Haskell: Concurrent

```
> t7 = do
>  v <- newMVar 0
```

Jetzt können wir diesen Speicherplatz an zwei Funktionen, die in unterschiedlichen Steuerfäden gestartet werden, übergeben.

Haskell: Concurrent

```
>  forkIO$ rep v "A" 1000000
>  rep v "B" 2000000
```

Die zentralen Funktionen auf MVar-Speicherplätzen sind:

- `takeMVar :: MVar a -> IO a`
  Diese Funktion liefert den in der Speicherstelle abgelegten Wert. Sollte die Speicherstelle allerdings leer sein, also noch kein Wert dort abgelegt sein, so wartet der Steuerfaden darauf, bis dort etwas von einem anderen Steuerfaden abgelegt wird. Die Funktion ist aber kein reines Lesen der Speicherplatzes, sondern nimmt den Wert richtiggehend aus der Variable heraus. Nach Ausführung der Funktion `takeMVar` ist die Speicherzelle leer.
- `putMVar :: MVar a -> a -> IO ()`
  Wie der Name erwarten lässt, legt diese Funktion Daten wieder in die Speicherzelle ab. Allerdings nur, wenn sie leer ist. Wenn da bereits ein Wert drin liegt, wartet der Steuerfaden, bis die Speicherzelle leer ist, also wieder Platz hat, um etwas hineinzulegen. Es wird also nicht ein eventueller Wert überschrieben.

Unsere bereits in der letzten Funktion aufgerufene Funktion `rep` wartet nun immer, bis in der übergebenen MVar ein Wert gefunden wird, holt diesen, erhöt ihn um 1 und legt das Ergebnis wieder in die Variable ab.

Haskell: Concurrent

```
> rep v x delay = forever$ do
>   s <- takeMVar v
>   putMVar v (s+1)
>   print  (x++show s)
>   threadDelay delay
```

Verklemmung Mit der Einführung von `MVar` können nun klassischen Verklemmungen (deadlocks) entstehen. Hierfür braucht es zwei Resourcen, die exklusiv zu verwenden sind. Diese lassen sich als zwei `MVar` darstellen. Und es braucht zwei Steuerfäden (threads), die diese zwei Resourcen benötigen. Die Verklemmung tritt auf, wenn sich die beiden Steuerfäden jeweils eine der beiden Resourcen exklusiv gesichert haben und nun darauf warten, dass die zweite Resource frei wird.

Es geht also um eine Funktion, die zwei Variablen braucht. Sie sichert sich die erste exklusiv, wartet eine Sekunde und sichert sich dann die zweite.

Haskell: Concurrent

```
> needsTwo v1 v2 xs = do
>   _ <- takeMVar v1
>   putStr$fromString xs
>   putStr$fromString ": I have one of the resources\n"
>   threadDelay 1000000
>   _ <- takeMVar v2
```

Anschließend wird eine Ausgabe gemacht und beide Variablen wieder frei gegeben. Der Inhalt der Variablen spielt dabei keine Rolle. Er ist in unserem Fall vom künstlichen Typ: `()`, der nur einen Wert hat, der auch als `()` notiert wird.

Haskell: Concurrent

```
>   print xs
>   putMVar v1 ()
>   putMVar v2 ()
```

Um eine einfache Verklemmung zu erzeugen, starten wir zwei Steuerfäden mit der Funktion `forkIO`, allerdings mit den beiden Variablen `v1` und `v2` jeweils in unterschiedlicher Reihenfolge. Es kommt zur Verklemmung.

Haskell: Concurrent

```
> deadlock = do
>  v1 <- newMVar ()
>  v2 <- newMVar ()
>  forkIO$ needsTwo v1 v2 "eins"
>  needsTwo v2 v1 "zwei"
```

Wenn wir dieses Programm starten, werden sich die beiden Steuerfäden jeweils eine der zwei Variablen sichern und damit leeren. Nun warten sie beide auf die Freigabe der zweiten Variable. Eine klassische Verklemmung, wie der Testaufruf zeigt.

Shell

```
panitz@px1:~/00021Concurrent$ ghci solution/Concurrent.lhs
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Concurrent       ( solution/Concurrent.lhs, interpreted )
Ok, one module loaded.
*Concurrent> deadlock
eins: I have one of the resources
zwei: I have one of the resources
*** Exception: thread blocked indefinitely in an MVar operation
*Concurrent>
```

Erstaunlicher Weise ist Haskell sogar in er Lage, nach geraumer Zeit die Verklemmung zu erkennen und das Programm abzubrechen. Dieses erkennt das Haskell Laufzeitsystem im Zuge der Garbage Collection.

MVars und Laziness Das Zusammenspiel der lazy Auswertung und den MVars ist nicht ganz ohne Fallstricke. Tatsächlich bewirkt ein Aufruf der Funktion `putMVar` nicht, dass ein fertig ausgwertetes Ergebnis in die Speicherzelle der MVar gelegt wird. Dieses kann man sich mit einem kleinen Test illustrieren. Hierzu brauchen wir eine Berechnung, die eine geisse Zeit braucht. Wir nehmen die Berechnung der Fibonaccizahl:

Haskell: Concurrent

```
> fib n
>  |n<2 = n
>  |otherwise = fib (n-2)+fib (n-1)
```

Die Berechnung von `fib 35` braucht ein paar Sekunden. Wir schreiben jetzt eine Funktion, die einen neuen Steuerfaden öffnet. Dieser soll in eine MVar ein Ergebnis legen. Dieses wird über ein Funktionsargument gemacht. Dann soll unsere Funktion darauf warten, dass der gestartete Steuerfaden sein Ergebnis in die Variable gelegt hat und dieses dann ausgeben.

Haskell: Concurrent

```
> tack function = do
>  v1 <- newEmptyMVar
>  forkIO (function v1)
>  putStr$fromString "Warte auf Fibonacci\n"
>  r <- (takeMVar v1)::IO Int
>  putStr$fromString "Variable mit Ergebnis ausgelesen\n"
>  putStr$fromString ("Ergebnis: ")
>  putStr$fromString$show r
>  putStr$fromString ("\nfertig!\n")
```

Die Funktion, die die Variable füllt, sei die Berechnung des Ausdrucks `(fib 35)`.

Haskell: Concurrent

```
> fib1 n v = do
>   putMVar v (fib n)
>   putStr$fromString "fib1 ist fertig\n"
```

Starten wir nun die Berehcnung mit dieser Funktion, kommt es zu einer zunächst unerwarteten Ausgabe, die insbesondere in ihrem Zeitverhalten verrät, wie Lazyness uns einen Strich durch die Rechnung macht:

Shell

```
*Concurrent> tack$ fib1 35
Warte auf Fibonacci
Variable mit Ergebnis ausgelesen
Ergebnis:
```

Mit dieser Ausgabe fängt das Warten an, bis ein paar Sekunden später der Rest auf der Kommandozeile erscheint:

Shell

```
 fib1 ist fertig
9227465
fertig!
```

Es wurde also die Variable `v1` schon längst ausgelesen und der gestartete Steuerfaden schon längst beendet, aber das Ergebnis der Berechnung lag noch lange nicht vor.

Der Grund hierfür ist, dass Haskell weiterhin die Auswertung von Ausdrücken so lang wie möglich verzögert. Es wird also gar nicht das Ergebnis von `fib 35` in die MVar gelegt, sondern lediglich der noch auszuwertende Ausdruck `fib 35`. Die eigentliche Auswertung findet dann gar nicht im abgezweigten Steuerfaden sondern im Hauptsteuerfaden statt.

Möchte man das nicht, so muss man im abgezweigten Steuerfaden die Auswertung erzwingen. Dieses kann man zum Beispiel durch den Aufruf der Funktion `evaluate` aus dem Modul `Control.Exception` erreichen.

Die Funktion für den zweiten Steuerfaden bekommt dann die Form.

Haskell: Concurrent

```
> fib2 n v = do
>   r <- evaluate$fib n
>   putMVar v  r
>   putStr$fromString "fib2 ist fertig\n"
```

Starten wir mit dieser Funktion den Steuerfaden, so bekommen wir den eigentlich erhofften Effekt, dass der abgezweigte Steuerfaden auch die Auswertung vornimmt.

Shell

```
*Concurrent> tack$ fib2 35
Warte auf Fibonacci
fib2 ist fertig
Variable mit Ergebnis ausgelesen
Ergebnis: 9227465
fertig!
```

Man sieht, es ist gar nicht so einfach, seine Arbeit in nebenläufigen Steuerfäden auszulagern.

Parallele Algorithem über MVar? Bisher haben wir nur über Nebenläufigkeit im Sinne, dass Berechnungen quasi gleichzeitig ablaufen gesprochen. Können wir dieses auch nutzen, um eine parallele Auswertung von Ausdrücken zu erzielen in dem Sinne, dass wir durch eine Mehrkernprozessortechnologie eine schnellere Auswertung bekommen?

Wir haben bereits in einem Steuerfaden die Berechnung von Fibonaccizahlen ausgelagert. Wenn wir den doppelten rekursiven Aufruf einmal in einen eigenen Stuerfaden auslagern und den anderen rekursiven Aufruf nebenläufig dazu im eigenen Steuerfaden durchführen, dann sollten wir eine Parallelisierung erzielt haben, vorausgesetzt das Betriebssystem kann die zwei Steuerfäden auf unterschiedlichen Prozessorkernen ausführen.

Versuchen wir es: eine Variante von `fib`, die für den einen rekursiven Aufruf einen eigenen Steuerfaden absplittet.

Haskell: Concurrent

```
> mvfib n
>  |n<=1 =return  n
>  |otherwise = do
>    v1 <- newEmptyMVar
>    forkIO (fib2 (n-1) v1)
>    x2 <- evaluate$fib (n-2)
>    x1 <- takeMVar v1
>    return (x1+x2)
```

Um für diese Funktion Laufzeittests durchzuühren, schreiben wir eine kleine `main`-Funktion.

Haskell: MVFib

```
import System.Environment(getArgs)
import Concurrent
main = do
  (inp1:_) <- getArgs
  r<-mvfib $read inp1
  print r
```

Um Parallelität auf mehreren Prozessorkernen nutzen zu können, ist das Programm mit dem `ghc` mit den Optionen `-threaded` und `-rtsopts` zu übersetzen. Erst so erhalten wir eine Applikation, die die Mehrkernarchitektur nutzen kann und der wir mit Kommandozeilenargumenten sagen können, wie viele Kerne bei der Ausführung verwendet werden können.

Shell

```
00021Concurrent$ ghc  -threaded -rtsopts Concurrent.lhs MVFib.hs
[1 of 2] Compiling Concurrent       ( Concurrent.lhs, Concurrent.o )
[2 of 2] Compiling Main             ( MVFib.hs, MVFib.o )
Linking MVFib ...
```

Jetzt können wir einmal schauen, wie das Laufzeitverhalten dieses Programms ist. Haskellprogramme sind zwar nun in der Lage, ein detailiertes Laufzeitverhalten zu protokollieren. Hierzu sind sie mit der Option `+RTS -f` zu starten. Der Einfachheit halber starten wir das Programm mit dem Unix `time` Kommando. Für die Berechenung von `(fib 42)` erhalten wir.

Shell

```
00021Concurrent$  time ./MVFib 42
fib2 ist fertig
267914296

real    0m43,364s
user    0m43,277s
sys     0m0,076s
```

Nun schauen wir, ob wir durch Ausnutzung der Parallelität etwas gewinnnen können. Hierzu geben wir als zusätzliches Argument dem Programm ein Argument für das Laufzeitsystem mit. Die Sektion von Laufzeitsystemargument beginnt mit `+RTS`. Wir sagen, dass zur Ausführung zwei Prozessorkerne verwendet werden können. Hierzu dient das Argument `-N2`.

Shell

```
00021Concurrent$  time ./MVFib 42 +RTS -N2
fib2 ist fertig
267914296

real    0m34,895s
user    0m55,897s
sys     0m0,859s
```

Wie man sieht, ist die Ausführung schneller geworden. Vielleicht nicht in dem erwarteteten Ausmaß, aber um gut ein Fünftel.

Paralleles Haskell Die im letzten Abschnitt durchgeführte Parallelisierung über eine synchronisierende Variable hat den Nachteil, dass wir uns in die IO-Monade begeben mussten.

Haskell bietet im Modul `Control.Parallel` zwei Funktionen an, die es ermöglichen Teilausdrücke paralllel auswerten zu lassen, ohne dass man selbst einen eigenen Steuerfaden absplittet und diesen über eine MVar synchronisiert. Die beiden Funktionen heißen `par` und `pseq`. Beide haben denselben Typ:

```
par :: a -> b -> b
pseq :: a -> b -> b
```

Beide Operatoren fungieren als eine Sequenz aus zwei Asudrücken. Der Wert ist der

Wert des zweiten Ausdrucks. Sie unterscheiden sich wieder im Punkt der Striktheit. Beide stoßen die Auswertung potentiell parallel an.

Haskell: Concurrent

```
> pfib d n
>  |n<=1 = n
>  |d<=0 = fib n
>  |otherwise
>    = let
>        x1 = fib (n-2)
>        x2 = pfib (d-1) (n-1)
>      in x1 `par` x2 `pseq` x1+x2
```

Auch hierfür machen wir ein kleines Testprogramm, mit dem wir wieder die Laufzeittests durchführen können, um zu sehen, ob es zu einer schnelleren Ausführung auf Mehrkernarchitekturen kommen kann.

Haskell: PFib

```
import System.Environment(getArgs)
import Concurrent
main = do
  (inp1:d:_) <- getArgs
  print$pfib (read d)$read inp1
```

Zunächst starten wir einmal das Programm ohne Zugriff auf mehrere Prozessorkerne.

Shell

```
00021Concurrent$  time ./PFib 42 4
267914296

real    0m46,114s
user    0m46,057s
sys     0m0,104s
```

Und dann ein weiteres Mal unter Ausnutzung von vier Prozessorkernen.

Shell

```
 00021Concurrent$  time ./PFib 42 4 +RTS -N4
267914296

real    0m30,792s
user    1m28,086s
sys     0m1,795s
```

Es läßt sich auch wieder eine Beschleunigung feststellen.

Die hier vorgestellte Parallelität wird als *semi-explizit* bezeichnet. Ziel ist es dabei, mit möglichst wenig Programmieranweisungen eine Parallelität in der Auswertung zu ermöglichen.

Wir wollen uns in diesem Lehrbrief weiter auf die Nebenläufigkeit konzentrieren und nicht die Parallelität vertiefen. Sehr ausführliche Folien zur Parallelität finden sich zur Lehrveranstaltung von David Sabel[Sab20].

1.3.2 Kanäle

Datenkanäle sind eine weitere Abstraktion, um zwischen verschiedenen Steuerfäden Daten zu kommunizieren und damit auch eine Synchronisation zu erreichen. Kanäle sind Datenströme für ein Elementtyp. In sie können Daten geschrieben und wieder ausgelesen werden. Gegenüber MVars haben sie den Vorteil, dass mehre Daten gleichzeitig in einem Kanal darauf warten, wieder ausgelesen zu werden. Datenkanäle sind intern mit MVars umgesetzt.

Eine zusätzliche Funktionalität für Kanäle ist, dass diese verdoppelt werden können. Wenn es mehrere Steuerfäden gibt, die aus einem Kanal Daten auslesen, so braucht nur einmal in den Kanal geschrieben werden, und beide Konsumenten sehen dann alle in den Kanal geschriebenen Daten.

So sind die vier entscheidenen Funktionen auf Kanälen für Erzeugen, Schreiben, Lesen und Duplizieren von Kanälen:

```
newChan   :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a
dupChan   :: Chan a -> IO (Chan a)
```

Als kleine erste Testanwenung sei eine Funktion geschrieben, die einen Kanal erzeugt und diesen verdoppelt. Den Orignal und das Duplikat werden zwei Steuerfäden übergeben, die aus dem Kanal lesen. Ein dritter Steuerfadeb schreibt in diesen Kanal Nachrichten.

Haskell: Concurrent

```
> chanT = do
>   c1 <- newChan
>   c2 <- dupChan c1
>   forkIO$readFrom "eins" c1
>   forkIO$readFrom "zwei" c2
>   writeTo 1 c1
```

Wann immer ein Element aus dem Kanal gelesen werden kann, schreiben die lesenden Steuerfäden dieses auf die Kommandozeile.

Haskell: Concurrent

```
> readFrom name c = forever$do
>   n <- readChan c
>   putStr$fromString  (name++": "++show n++"\n")
```

Der schreibene Steuerfaden schreibt jede Sekunde eine neue Zahl in den Kanal.

Haskell: Concurrent

```
> writeTo n c = do
>  writeChan c n
>  threadDelay 1000000
>  writeTo (n+1) c
```

Startet man die Funktion, so sieht man dass beide Leder alle Inhalte bekommen.

Shell

```
*Concurrent> chanT
zwei: 1
eins: 1
eins: 2
zwei: 2
```

2 Beispielanwendung mit TCP Socket

Als kleine Beispielanwendung für Nebenläufigkeit soll ein kleiner Chatserver dienen, der ein Socket verwendet und auf diesem Verbindungen akzeptiert. Hierzu verwenden wir das Modul `Network.Socket`, das die benötigten Funktionen bereitstellt, um Sockets zu öffnen.

Zunächst öffnen wir ein Socket für das Internet Protocol Version 4 mit einem Strom zur Kommunikation. Diesen binden wir an den Port 4242 unseres lokalen Rechners und warten dort auf Verbindungen.

Haskell: Concurrent

```
> startChatServer = do
>   sock <- socket AF_INET Stream defaultProtocol
>   bind sock (SockAddrInet 4242 (tupleToHostAddress (127,0,0,1)))
>   listen sock 2
```

Jetzt kümmern wir uns um die Nebenläufigkeit. Wir akzeptieren beliebige Verbindungen auf unserem Socket, die wir dann nebenläufig im Dialog bedienen. Alle Verbindungen teilen sich dann einen Kanal, in dem Strings geschrieben werden.

Haskell: Concurrent

```
>   chan <- newChan
>   forever$ do
>     con <- accept sock
>     forkIO (verbindung con chan)
```

Wenn eine Verbindung zu unserem Server aufgemacht wurde, holen wir uns den Kommunikationsstream für die Kommunikation. Dieses ist ein Handle, auf dem es die Funktionen `hPutStrLn` und `hGetLine` gibt.

Haskell: Concurrent

```
> verbindung (sock, _) chan = do
>    hdl <- socketToHandle sock ReadWriteMode
```

Wir lassen uns vom neuen Chatteilnehmer eine Namen geben:

Haskell: Concurrent

```
>    hPutStrLn hdl "Bitte einen Chatnamen eingeben?"
>    name <- hGetLine hdl
```

Über den Kanal informieren wir alle Chatteilnehmer über den neuen Teilnehmer:

Haskell: Concurrent

```
>    writeChan chan ("> " ++ (init name) ++ " ist neu imm Chat.")
```

Auch für den neuen Teilnehmer wird der Kanal dupliziert:

Haskell: Concurrent

```
>    commLine <- dupChan chan
```

Wir brauchen nun zwei nebenläufig unendlich laufende Prozesse. Einer der immer schaut, ob im Kanal neue Nachrichten eingetragen wurden und diese dann über den Socket verschickt.

Haskell: Concurrent

```
>    forkIO $ forever$ do
>      line <- readChan commLine
>      hPutStrLn hdl line
```

Und einer der schaut, ob über den Kanal des Sockets neue Nachrichten eingegeben wurden und diese dann an alle Chatteilnehmer über den gemeinsamen Kanal schickt.

Haskell: Concurrent

```
>    forever$ do
>      line <- hGetLine hdl
>      writeChan chan ((init name) ++ ": " ++ line)
```

Damit haben wir in etwa 20 Zeilen einen kleinen Chatserver umgesetzt.

Wir können diesen einmal starten.

Shell

```
00021Concurrent$ ghci solution/Concurrent.lhs
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Concurrent       ( solution/Concurrent.lhs, interpreted )
Ok, one module loaded.
*Concurrent> startChatServer
```

Wir haben keinen eigenen Client für unseren Chatserver in Haskell geschrieben, weil wir mit dem Programm `telnet` die Verbindung öffnen können.

Shell

```
panitz@px1:~$ telnet localhost 4242
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Bitte einen Chatnamen eingeben?
Bernd
--> Klaus ist neu imm Chat.
hallo Klaus
Bernd: hallo Klaus
Klaus: Hallo Bernd
```

Dazu haben wir eine zweiten Client in einer telnet-Session geöffnet.

Shell

```
panitz@px1:~$ telnet localhost 4242
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Bitte einen Chatnamen eingeben?
Klaus
Bernd: hallo Klaus
Hallo Bernd
Klaus: Hallo Bernd
```

Wie man sieht, ist es tatsächlich ein kleiner, funktionaler Chat-Server.

3 Lernzuwachs

- Threads
- Globale Variablen
- Parallelisierung

Literatur

[JGF96] Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. Concurrent haskell. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 295–308. ACM Press, 1996.

[Sab20] Sabel, David. Parallelität und Nebenläufigkeit in Haskell. `www.tcs.ifi.lmu.de/lehre/ss-2020/fun/material/folien/folien-10-druckversion/view`, 2020. [Online; accessed 8-May-2021].