Zweiter Haskell-Workshop
im OpenLab Augsburg
github.com/curry-club-aux/haskell-workshop

Curry-Club AUGSBURG
:: ((a, u) → x) → a → u → x

Übungsblatt zu Haskell

Learn You a Haskell for Great Good!

1 Statistisch aussagekräftige Benchmarks

Die Haskell-Community liebt die Bibliothek *Criterion*, um die Laufzeit von Haskell-Programmen zu messen. Diese macht viel mehr, als nur gegebenen Code mehrmals auszuführen und dann die durchschnittliche Laufzeit zu berechnen. Sie bestimmt auch die Standardabweichung der Laufzeit und gibt ein statistisches Maß für die Verlässlichkeit der geschätzten Werte aus.

Ihre Benutzung ist kinderleicht:

1. `stack install criterion`
2. Folgende Vorlage anpassen:

```haskell
import Criterion.Main

fib :: Integer -> Integer
fib = ...

main = defaultMain
    [ bgroup "fib"
        [ bench "10" $ whnf fib 10
        , bench "20" $ whnf fib 20
        , bench "30" $ whnf fib 30
        ]
    ]
```

3. Das Programm ausführen. Wenn man dabei die Option `--output foo.html` übergibt, erstellt die Criterion-Bibliothek eine interaktive HTML-Seite, der man unter anderem die Verteilung der Messwerte entnehmen kann.

2 Erste Schritte mit Nebenläufigkeit

Da es in Haskell keinen veränderlichen Zustand gibt, können Haskell-Ausdrücke in erster Näherung in beliebiger Reihenfolge und auf beliebigen Prozessorkernen ausgewertet werden. GHC verteilt aber nicht von selbst Aufgaben auf mehrere Kerne.

Es gibt in Haskell vier verschiedene Möglichkeiten, Nebenläufigkeit zu erreichen, die man je nach Anwendungszweck einsetzen kann.

- Parallelisierungsannotationen. Puren Code kann man einfach und ohne Umstrukturierung des Programms mit Auswertungsannotationen versehen, wie zum Beispiel „führe das folgende `map` parallel aus“ oder „falls Ressourcen vorhanden sind, beginne die Auswertung des folgenden Ausdrucks im Hintergrund“.

- Threads. Wie in anderen Sprachen auch kann man explizit Threads erstellen. Threads können auf diverse Weisen miteinander kommunizieren, zum Beispiel mittels gemeinsamer veränderlicher Variablen (`MVar`) und Channels (`Chan`). Dieser recht explizite Zugang zu Nebenläufigkeit ist also sehr ähnlich zum Zugang von anderen Sprachen wie Python oder JavaScript mit Node.js. Anders als in diesen Sprachen gibt es aber keine „Callback-Hölle“.
- Shared Transactional Memory (STM). Parallelisierungsannotationen helfen nicht bei Code, der Nebenwirkungen verursachen muss. Wenn man aber auf Threads zurückgreifen würde, müsste man wie in anderen Sprachen auch auf korrektes Locking und Race Conditions achten; das ist mühsam und fehleranfällig. STM ist eine Technik, mit der man vorgeben kann, dass speziell gekennzeichneter Code so abläuft, als wäre das Programm rein sequenziell geschrieben. Der große Vorteil an STM ist *Kompositionalität*: Man kann Code rein lokal verstehen und kombinieren, ohne auf Auswirkungen von parallel ablaufenden Programmteilen achten zu müssen.
- Data Parallel Haskell (DPH). Dabei kümmern sich der Compiler und die Laufzeitumgebung selbstständig um eine effiziente Verteilung des auszuwertenden Codes. DPH ist ein Forschungsprojekt, das noch nicht seinen Weg in die aktuelle GHC-Version gefunden hat.

2.1 Parallelisierungsannotationen

Aus dem ersten Workshop ist ja die Funktion `seq :: a -> b -> b` bekannt. Wird der Ausdruck `seq x y` ausgewertet, so wird zunächst `x` ausgewertet, das Ergebnis verworfen, und dann `y` zurückgegeben.

Ein Aufruf wie `seq 42 y` ist nicht besonders sinnvoll. Wenn aber die Auswertung von `x` die Auswertung von Teilen einer Datenstruktur anstößt, so bleiben die Ergebnisse gespeichert. Die folgende GHCi-Sitzung illustriert das:

```
> let x = fib 30  -- kehrt sofort zurück
> seq x "Hallo"   -- dauert lange, da ‘x‘ ausgewertet wird
"Hallo"
> x               -- kehrt sofort zurück
832040
```

Nun gibt es neben `seq` auch die Funktion `par :: a -> b -> b` aus dem Modul `Control.Parallel` (aus dem Paket `parallel`). Semantisch ist `par x y` identisch zu `y`. Als Nebenwirkung wird aber ein *Spark* erzeugt, der `x` im Hintergrund parallel auswertet.

In GHCi sieht das zum Beispiel so aus:

```
> let fib :: Int -> Int; fib n = if n <= 1 then n else fib (n-1) + fib (n-2)
> let x = fib 30
> let y = fib 30
> (x,y)
(832040,832040)   -- die beiden Komponenten werden nacheinander
                  -- berechnet und ausgegeben

> import Control.Parallel
> let a = fib 30
> let b = fib 30
> b ‘par‘ (a,b)
(832040,832040)   -- nach anfänglicher Verzögerung werden beide
                  -- Komponenten in einem Rutsch ausgegeben
```

Wichtig: Standardmäßig verwendet die Laufzeitumgebung nur einen einzigen Betriebssystem-Thread. Damit können keine Sparks im Hintergrund ausgeführt werden. Man muss seinen Code mit der Option `-threaded` kompilieren und beim Ausführen dem Laufzeitsystem mitteilen, dass es mehrere Betriebssystem-Threads verwenden soll:

```
# Kompilieren mit:
$ ghc --make -O2 -threaded Main

# Ausführen mit:
$ ./Main +RTS -N4 -RTS   # genau vier Betriebssystem-Threads verwenden
$ ./Main +RTS -N -RTS    # sinnvolle Anzahl Betriebssystem-Threads verwenden

# Interaktive Shell:
$ ghci +RTS -N -RTS
```

Aufgabe 1. *Was bedeutet eigentlich Auswertung?*

Erkläre, wieso in folgender GHCi-Sitzung scheinbar `b` *nicht* im Hintergrund ausgewertet wird. Denke daran, GHCi mit der Option `+RTS -N -RTS` zu starten.

```
> import Control.Parallel
> let a = [fib 30]
> let b = [fib 30]
> b `par` (a,b)
([832040],[832040])
```

Aufgabe 2. *Paralleles Map*

Schreibe eine Funktion `parMap :: (a -> b) -> [a] -> [b]`, die semantisch identisch zu `map` ist, aber alle Werte parallel berechnet.

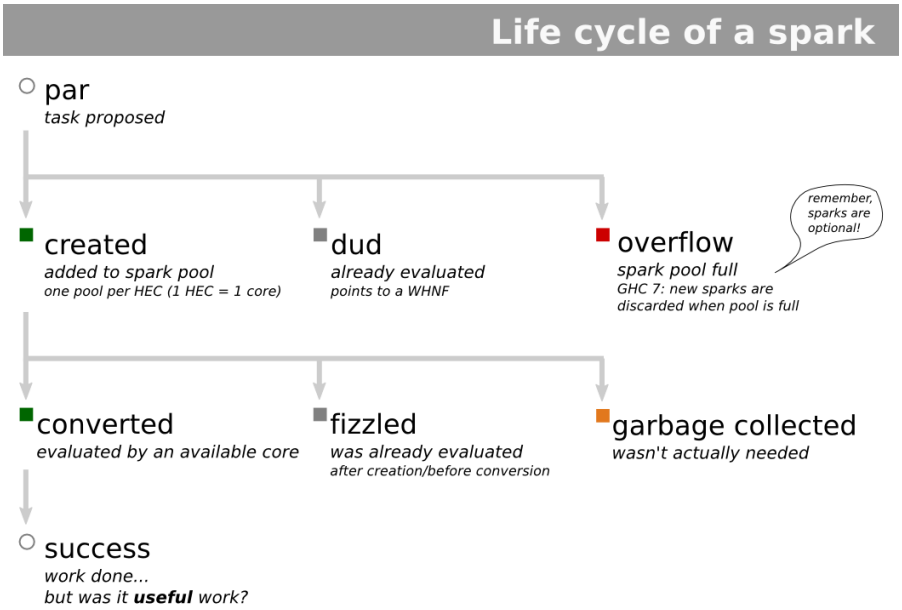
Auf einem Mehrkern-Computer sollte also

```
> parMap fib [30,30,30,30]
```

deutlich schneller ablaufen als `map fib [30,30,30,30]`.

Zu Sparks ist noch viel mehr zu sagen. An dieser Stelle nur zwei Bemerkungen: Startet man sein Programm mit den Optionen `+RTS -N -s -RTS`, so werden nach Beendigung Statistiken ausgegeben. Diese beinhalten unter anderem, wie viele Sparks erzeugt wurden und wie viele *fizzelten* – das heißt, dass der zu berechnende Wert schon vom Hauptthread angefordert wurde, noch bevor der Spark loslegen konnte.

Außerdem gibt es *ThreadScope*, mit dem die Auslastung durch Threads und Sparks visualisiert werden kann.



Der Lebenszyklus eines Sparks. Quelle:
`https://wiki.haskell.org/ThreadScope_Tour/Spark`

2.2 Threads

Mit `forkIO :: IO a -> IO ThreadId` aus dem Modul `Control.Concurrent` erzeugt man einen *leichtgewichtigen Thread*. Die übergebene IO-Aktion wird in diesem Thread ausgeführt; Rückgabewert ist ein Wert vom Typ `ThreadId`, mit dem man den Thread noch nachträglich kontrollieren (etwa vorzeitig beenden) kann.

Die Laufzeitumgebung kommt mit sehr vielen – Millionen – von leichtgewichtigen Threads klar. Sie werden auf eine kleine Anzahl echter Threads auf Betriebssystem-Level verteilt.

In speziellen Anwendungsfällen ist es nötig, Betriebssystem-Threads statt leichtgewichtiger Threads zu erzeugen. Das ist mittels der Funktion `forkOS :: IO a -> IO ThreadId` ebenfalls möglich.

Wichtig: Wenn man Threads nur verwenden möchte, um mit simultan stattfindenden IO-Aktionen umzugehen (etwa Anforderungen mehrerer gleichzeitig verbundener Clients über das Netzwerk entgegennehmen), genügt prinzipiell ein einzelner Betriebssystem-Thread. Wenn man aber mit Threads tatsächlich auch mehrere Berechnungen parallel ausführen möchte, muss man wie im vorherigen Abschnitt beschrieben seinen Code mit der Option `-threaded` kompilieren und beim Ausführen dem Laufzeitsystem mit `+RTS -N -RTS` mitteilen, mehrere Betriebssystem-Threads zu verwenden.

Aufgabe 3. *Hallo Welt aus zwei Threads*

Schreibe ein Haskell-Programm, das einen leichtgewichtigen Thread erzeugt und den ausführenden Lambroiden sowohl vom Hauptthread als auch dem erzeugten Thread mit `putStrLn` grüßt.

Aufgabe 4. *Vermischte Ausgabe*

Schreibe ein Haskell-Programm, das zwei leichtgewichtige Threads erzeugt. Der eine Thread soll tausendmal das Zeichen `'a'` ausgeben, der andere das Zeichen `'b'`. Was passiert?

Aufgabe 5. *Sleep Sort*

Implementiere *Sleep Sort*: Erzeuge für jedes Element `x` einer gegebenen Liste von (kleinen) natürlichen Zahlen einen Thread, der sich gleich nach seiner Erstellung für eine zu `x` proportionale Zeit schlafen legt und anschließend `x` auf dem Terminal ausgibt.

Tipp. Verwende die Funktion `threadDelay :: Int -> IO ()`, die den momentan laufenden Thread für eine gegebene Anzahl Mikrosekunden schlafen legt.

Eine primitive Möglichkeit der Kommunikation zwischen Threads sind (thread-sichere) veränderliche Variablen. Eine solche kann zu jedem Zeitpunkt leer sein oder einen Wert enthalten. Man erstellt sie mit `newEmptyMVar :: IO (MVar a)` oder, wenn man die Variable gleich initialiseren möchte, mit `newMVar :: a -> IO (MVar a)`.

Mit `readMVar :: MVar a -> IO a` holt man den aktuellen Wert einer übergebenen Variable. Sollte die Variable leer sein, blockiert dieser Aufruf so lange, bis die Variable durch einen anderen Thread gefüllt wird.

Eine Variante ist die Funktion `takeMVar :: MVar a -> IO a`, die nach Auslesen der Variable diese leert.

Mit `putMVar :: MVar a -> a -> IO ()` setzt man den Inhalt einer Variable. Wenn diese zum Zeitpunkt des Aufrufs nicht leer sein sollte, wird der vorhandene Inhalt nicht überschrieben. Stattdessen wird der ausführende Thread so schlafen gelegt, bis ein anderer Thread die Variable mit `takeMVar` leert. (Es gibt auch `tryPutMVar :: MVar -> a -> IO Bool`, das den Thread nicht schlafen geht und den Erfolg durch den Rückgabewert anzeigt.)

Aufgabe 6. *Lesen aus einer dauerhaft leeren Variable*

Was macht folgender Code? Wie reagiert das Laufzeitsystem von GHC?

```
import Control.Concurrent

main = do
    ref <- newEmptyMVar
    takeMVar ref
```

Aufgabe 7. *Ein einfaches Beispiel zu Variablen*

Schreibe ein Programm, das zwei leichtgewichtigen Threads erzeugt, die je eine große Fibonacci-Zahl berechnen und das Ergebnis in je einer Variable speichern. Der Hauptthread soll dann die beiden Ergebnisse ausgeben.

Aufgabe 8. *Vorsicht vor Deadlocks*

Was macht folgender Code? Wie reagiert das Laufzeitsystem von GHC?

```
import Control.Concurrent

main = do
    ref1 <- newEmptyMVar
    ref2 <- newEmptyMVar
    forkIO $ takeMVar ref2 >> putMVar ref1 "Hallo Welt"
    putStrLn =<< takeMVar ref1
```

Aufgabe 9. *Warten auf Kinder*

Oft möchte man im Hauptthread die Beendigung gestarteter Threads abwarten. Das ist zum Beispiel mit folgendem Code möglich (den es natürlich auch schon in verpackter Form im Modul `Control.Concurrent.Async` gibt). Vollziehe ihn nach!

```haskell
import Control.Monad
import Control.Concurrent

forkThread :: IO () -> IO (MVar ())
forkThread proc = do
    ref <- newEmptyMVar
    forkFinally proc $ \_ -> putMVar ref ()
    return ref

main = do
    jobs <- mapM forkThread [...]
    mapM_ takeMVar jobs
```

Neben veränderlichen Variablen gibt es noch *Kanäle* zur Kommunikation zwischen Threads. Kanäle können anders als Variablen mehr als einen Wert zwischenspeichern. Man erzeugt einen Kanal mit `newChan :: IO (Chan a)`, pusht einen Wert durch `writeChan :: Chan a -> a -> IO ()` und poppt den vordersten Wert mit `readChan :: Chan a -> IO a`. Der Aufruf von `readChan` blockiert, falls der Kanal leer ist.

Aufgabe 10. *Sleep Sort kanalbasiert*

Modifiziere deinen Sleep-Sort-Algorithmus derart, dass die sortierten Werte nicht auf dem Terminal ausgegeben, sondern in einen Kanal geschrieben werden. Dieser soll dann in einem Rutsch ausgegeben werden.

Zum Ende dieses Abschnitts sei bemerkt, dass man selten auf der Ebene dieser Aufgaben programmieren muss. Für viele Einsatzgebiete gibt es schon fertige Kombinatoren-Bibliotheken zum nebenläufigen Programmieren.

Aufgabe 11. *Projekt: Ein einfacher Chat-Server*

Vervollständige folgende Vorlage für einen einfachen Chat-Server. Clients sollen sich mit ihm auf TCP-Port 4242 verbinden können. Eingehende Nachrichten sollen an alle verbundenen Clients weitergeleitet werden.

Diese Vorlage ist auf einem niedrigen Level, mit expliziten Socket-Operationen, geschrieben. Normalerweise würde man eine High-Level-Streaming-Bibliothek wie Conduits oder Pipes verwenden. Diese kümmern sich auch automatisch um ordnungsgemäßes Abmelden von Clients.

Tipp. Verwende die Funktion `dupChan :: Chan a -> IO (Chan a)`. *Bonusaufgabe.* Identifiziere das Speicherleckproblem und löse es.

```haskell
module Main where

import Control.Monad
import Control.Concurrent
import Network.Socket
import System.IO

main :: IO ()
main = do
    -- Lausche auf Port 4242.
    sock <- socket AF_INET Stream 0
    setSocketOption sock ReuseAddr 1
    bindSocket sock (SockAddrInet 4242 iNADDR_ANY)
    listen sock 10
```

```
    -- Setze einen Kanal auf. Was in diesen Kanal geschrieben wird,
    -- soll an alle verbundenen Clients weitergeleitet werden.
    ...

    -- Die Hauptschleife: Akzeptiere laufend neue Verbindungen und
    -- bearbeite sie.
    forever $ do
        (conn,_) <- accept sock
        hdl      <- socketToHandle conn ReadWriteMode
        hSetBuffering hdl NoBuffering
        -- `hdl` ist nun ein gewöhnlicher Handle, mit dem `hGetLine`
        -- und `hPutStrLn` verwendet werden können.

        -- Dupliziere den Kanal, um mehrere Zuhörer zu unterstützen.
        ...

        -- Schreibe gelesene Nachrichten in den Kanal.
        forkIO $ forever $ do
            msg <- hGetLine hdl
            ...

        -- Leite Nachrichten der anderen Verbindungen weiter.
        forkIO $ forever $ do
            ...
            hPutStrLn hdl msg
```

2.3 Shared Transactional Memory (STM)

In dem folgenden Programm kommt es zu einer Race Condition. Wiederholte Aufrufe des Programms werden verschiedene Ergebnisse liefern.

```
import Control.Concurrent
import Control.Monad

forkThread :: IO () -> IO (MVar ())
forkThread = {- siehe oben -}

go :: IORef Integer -> IORef Integer -> IO ()
go xRef yRef = do
    x <- readIORef xRef
    y <- readIORef yRef
    let x' = y  + 1
        y' = x' + 1
    writeIORef xRef x'
    writeIORef yRef y'

main = do
    xRef <- newIORef 1
    yRef <- newIORef 2
    jobs <- replicateM 40000 $ forkThread $ go xRef yRef
    mapM_ takeMVar jobs
```

```
    x <- readIORef xRef
    y <- readIORef yRef
    print (x, y)
```

Mit STM passiert das nicht. Statt `IORef`'s verwendet man dann `TVar`'s. Die Operationen übertragen sich wörtlich, spielen sich dann aber in der `STM`- statt der `IO`-Monade ab: `newTVar :: a -> STM (TVar a)` und so weiter. Man führt STM-Aktionen mit `atomically :: STM a -> IO a` aus.

Der angepasste Code sieht so aus:

```
import Control.Concurrent
import Control.Concurrent.STM
import Control.Monad

forkThread :: IO () -> IO (MVar ())
forkThread = {- siehe oben -}

go :: TVar Integer -> TVar Integer -> STM ()
go xRef yRef = do
    x <- readTVar xRef
    y <- readTVar yRef
    let x' = y  + 2
        y' = x' + 2
    writeTVar xRef x'
    writeTVar yRef y'

main = do
    xRef <- newTVarIO 1
    yRef <- newTVarIO 2
    jobs <- replicateM 40000 $ forkThread $ atomically $ go xRef yRef
    mapM_ takeMVar jobs
    x <- readTVarIO xRef
    y <- readTVarIO yRef
    print (x, y)
```

Wie funktioniert STM? In erster Näherung so: Wird ein `atomically`-Block ausgeführt, so werden Änderungen an `TVar`'s nicht sofort geschrieben. Stattdessen werden sie in einem Log notiert. Am Ende des Blocks prüft das Laufzeitsystem in einer atomaren Operation, ob sich seit Ablauf des Blocks seine Abhängigkeiten (zum Beispiel veränderliche Variablen, auf die lesend zugegriffen wurde) geändert haben. Wenn nein, macht es die Änderungen an den `TVar`'s wirksam. Wenn ja, wird der Block einfach erneut ausgeführt. Da in der STM-Monade nicht beliebige Nebenwirkungen wie `fireMissiles` möglich sind, ist das ein fundiertes Vorgehen.