Übungsblatt zu Haskell

Learn You a Haskell for Great Good!

1 Mehr zu Monaden

Aufgabe 1. *Umgekehrte polnische Notation*

In der umgekehrten polnischen Notation schreibt man einen mathematischen Ausdruck wie

$$(1 + 2) \cdot 3 - 4 \cdot 5$$

so:

```
1 2 + 3 * 4 5 * -
```

Der Vorteil an dieser Notation ist, dass man keine Klammern und keine Regeln für die Präzedenz von Operatoren („Punkt vor Strich“) benötigt. Die Auswertung eines Ausdrucks in umgekehrter polnischer Notation erfolgt mit einem Stack. Konstanten pushen jeweils ihren Wert auf den Stack. Rechenoperationen poppen die obersten zwei Werte des Stacks und pushen das Ergebnis.

Wir formulieren das in Haskell wie folgt:

```haskell
data Instr a
    = Lit a
    | Add
    | Sub
    | Mult
    | Div
    deriving (Show,Eq)

type Stack a = [a]

ex :: [Instr Double]
ex = [Lit 1, Lit 2, Add, Lit 3, Mult, Lit 4, Lit 5, Mult, Sub]
```

a) Schreibe eine Funktion `exec :: (Num a, Fractional a) => Instr a -> Stack a -> Stack a`, die eine Instruktion als Argument nimmt und diese ausführt: also einen gegegeben Stack in einen neuen transformiert. Im Fehlerfall (Division durch Null, zu wenige Argumente auf dem Stack) soll die Funktion einfach mit `error` terminieren.

b) Schreibe die Funktion nun unter Verwendung der State-Monade um. Der neue Typ soll also `exec' :: (Num a, Fractional a) => Instr a -> State (Stack a) ()` sein. Denk daran, das Modul `Control.Monad.State` zu importieren. Schreibe und verwende Hilfsfunktionen `push :: a -> State (Stack a) ()` und `pop :: State (Stack a) a`. Ein Fall in der Definition von `exec'` soll so aussehen:

```
exec' Add = do
    x <- pop
    y <- pop
    push (x + y)
```

c) Kombiniere die Funktion `mapM_ :: (Monad m) => (a -> m b) -> [a] -> m [b]` mit deiner Funktion `exec'` und der Funktion `execState` aus `Control.Monad.State`, um einen Interpreter für Ausdrücke in umgekehrter polnischer Notation zu erhalten: eine Funktion vom Typ `(Num a) => [Instr a] -> a`. (Zurückgegeben werden soll das oberste Element des Stacks.)

Aufgabe 2. *Vermeidung von unübersichtlichen Fehlerbehandlungen I*

a) Implementiere zum Aufwärmen folgende drei Funktionen.

```
safeHead :: [a] -> Maybe a
safeLast :: [a] -> Maybe a
safeTail :: [a] -> Maybe [a]
```

b) Folgender Code verwendet diese Funktionen, um das erste Element einer gegebenen Liste mit dem zweiten zu multiplizieren. Bewundere, wie unübersichtlich der Code durch die verschachtelte Fehlerbehandlung ist.

```
ex :: (Num a) => [a] -> Maybe a
ex xs = case safeHead xs of
    Nothing -> Nothing
    Just x  -> case safeTail xs of
        Nothing  -> Nothing
        Just xs' -> case safeHead xs' of
            Nothing -> Nothing
            Just y  -> Just (x * y)
```

c) Schreibe den Code nun mit Hilfe der do-Notation in der Maybe-Monade um! Er soll dann keine Fallunterscheidungen mehr enthalten und etwa so aussehen:

```
ex :: (Num a) => [a] -> Maybe a
ex xs = do
    ...   -- hier irgendwas mit safeHead
    ...   -- hier irgendwas mit safeTail
    ...   -- hier irgendwas mit safeHead
    ...   -- hier irgendwas mit x * y
```

d) Schreibe eine generische Funktion `iterateM :: (Monad m) => Int -> (a -> m a) -> a -> m a`.

e) Kombiniere `safeHead`, `safeTail` und `iterateM` zu einer Funktion `safeNth :: [a] -> Int -> Maybe a`.

Aufgabe 3. *Vermeidung von unübersichtlichen Fehlerbehandlungen II*

Unsere übersichtliche Lösung aus der vorherigen Aufgabe hat ein Manko: Im Fehlerfall gibt sie keine Fehlermeldungen aus. Diesen Missstand können wir beheben, indem wir nicht die `Maybe`-, sondern die `Either String`-Monade einsetzen.

a) Ändere die Funktionen aus Teilaufgabe a) der vorherigen Aufgabe so ab, dass sie im Fehlerfall mit `Left` eine Fehlermeldung zurückgeben. Ihre Typen sollen sein:

```
safeHead :: [a] -> Either String a
safeLast :: [a] -> Either String a
safeTail :: [a] -> Either String [a]
```

b) Ändere analog die Funktion `ex` aus der zweiten Teilaufgabe ab. Ihr neuer Typ soll `(Num a) => [a] -> Either String a` sein. Spüre den dabei entstehenden Schmerz.

c) Bewundere, wie sich die Lösung aus Teilaufgabe c) der vorherigen Aufgabe mühelos auf die neue Situation überträgt. Nur die (ohnehin optionale) Typsignatur muss angepasst werden!

Aufgabe 4. *Logging mit der Writer-Monade*

Aufgabe 5. *Implementierung der Maybe-Monade*

Implementiere die Monaden-Instanz von `Maybe` neu. Vervollständige also folgendes Programm:

```
-- kein ‘import Data.Maybe‘ hier!

data Maybe a = Nothing | Just a deriving (Show,Eq)

instance Functor Maybe where
    fmap f Nothing  = ...
    fmap f (Just x) = ...

instance Applicative Maybe where
    pure x = ...
    Nothing <*> _       = ...
    Just f  <*> Nothing = ...
    Just f  <*> Just x  = ...

instance Monad Maybe where
    Nothing >>= g = ...
    Just x  >>= g = ...
```

Aufgabe 6. *Ein Brainteaser zur Reader-Monade*

Was macht folgende Funktion `f`? Und wieso?

```
f :: [a] -> a
f = head ‘fmap‘ reverse
```

Aufgabe 7. *Ein erster Monadenturm*

Wir haben gesehen, dass man mit der State-Monade veränderlichen Zustand durchfädeln kann. Mit der Funktion

`runState :: State s a -> s -> (a,s)`

kann man eine State-Aktion ausführen. Außerdem haben wir gesehen, dass man mit der Maybe-Monade Fehlerfälle übersichtlicher behandeln kann. Gibt es eine Möglichkeit, diese beiden Fähigkeiten zu kombinieren?

Ja, die gibt es! Und zwar baut man sich den *Monadenturm* `StateT s Maybe`. Man verlässt diese Monade mit der Funktion

`runStateT :: StateT s Maybe a -> Maybe (a, s)`.

Dabei ist `StateT s` ein *Monadentransformer*: Gegeben eine Monade `M`, so ist `StateT s M` eine neue Monade, die die Fähigkeiten von `M` sowie veränderlichen Zustand vom Typ `s` unterstützt.

Schreibe die Funktion `exec' :: (Num a, Fractional a) => Instr a -> State (Stack a) ()` aus der Aufgabe zur umgekehrten polnischen Notation so um, dass im Fehlerfall nicht mit `error` eine Ausnahme geworfen wird, sondern `Nothing` zurückgegeben wird. Der neue Typ soll also `exec'' :: (Num a, Fractional a) => Instr a -> StateT (Stack a) Maybe ()` sein.

Tipp. Wegen Typklassenmagie kannst du weiterhin die Funktionen `get` und `put` verwenden. Diese haben hier die Typen `get :: StateT (Stack a) Maybe (Stack a)` und `put :: Stack a -> StateT (Stack a) Maybe ()`. Mit der Funktion `lift :: Maybe r -> StateT (Stack a) Maybe r` kannst du Werte aus der Basismonade, in unserem Fall also `Maybe`, in die transformierte Monade heben. Nützlich wird vor allem `lift Nothing :: StateT (Stack a) Maybe` sein.

Aufgabe 8. *Eine Monade für Wahrscheinlichkeiten*

Wenn du dich für Statistik oder Wahrscheinlichkeitstheorie interessiert, dann schau dir `http://www.randomhacks.net/2007/02/22/bayes-rule-and-drug-tests/` und `http://www.randomhacks.net/2007/02/21/refactoring-probability-distributions/` an.

Damit kann man Code wie den folgenden schreiben:

```
die :: Dist Int
die = uniform [1..6]

twoDice :: Dist (Int,Int)
twoDice = do
    x <- die
    y <- die
    return (x,y)
-- kürzer: twoDice = liftM2 (,) die die
-- oder:   twoDice = (,) <$> die <*> die
-- (Mit der in Arbeit befindlichen Erweiterung "Idiom Brackets"
-- wird die Notation noch verschönert werden!)

sumOfTwoDice :: Dist Int
sumOfTwoDice = do
    x <- die
    y <- die
    return $ x + y
-- oder: sumOfTwoDice = (+) <$> die <$> die
```

Aufgabe 9. *Für Unerschrockene: Die Continuation-Monade*

XXX: Listen-Monade!

2 QuickCheck

Eine der coolsten Haskell-Bibliothek überhaupt ist *QuickCheck*. Mit QuickCheck ist es supereinfach, geschriebene Funktionen auf ihre Korrektheit zu testen. Während in anderen Programmiersprachen ein Test oft bloß aus einer handvoll selbst ausgewählter Beispieleingaben und den zugehörigen erwarteten Ausgaben besteht, kann man mit

QuickCheck erwartete *Eigenschaften* von Funktionen formulieren und überprüfen lassen. Dadurch deckt man häufig Fehler in Sonderfällen auf, an die man sonst gar nicht gedacht hätte.

Als Beispiel wollen wir ein paar Eigenschaften der Listenverkettung `(++) :: [a] -> [a] -> [a]` formulieren und checken:

```
import Data.List (isPrefixOf, isSuffixOf)

-- Die Länge der Verkettung zweier Listen ist die Summe der Längen.
prop_concat_length :: [Int] -> [Int] -> Bool
prop_concat_length as bs = length (as ++ bs) == length as + length bs

-- Die Liste ‘as‘ ist ein Präfix von ‘as ++ bs‘
prop_concat_prefix :: [Int] -> [Int] -> Bool
prop_concat_prefix as bs = as ‘isPrefixOf‘ as ++ bs
-- bzw. take (length as) (as ++ bs) == as

-- Die Liste ‘bs‘ ist ein Suffix von ‘as ++ bs‘
prop_concat_suffix :: [Int] -> [Int] -> Bool
prop_concat_suffix as bs = bs ‘isSuffixOf‘ as ++ bs
```

Jede Eigenschaft ist eine Funktion, die ein paar Eingaben nimmt und dann zurückgibt, ob die Eigenschaft für diese Eingaben erfüllt ist. Wir können nun diese Eigenschaften mit der `quickCheck`-Funktion in GHCi überprüfen:

```
Prelude> :m +Test.QuickCheck
Prelude> quickCheck prop_concat_length
+++ OK, passed 100 tests.
Prelude> quickCheck prop_concat_prefix
+++ OK, passed 100 tests.
Prelude> quickCheck prop_concat_suffix
+++ OK, passed 100 tests.
```

QuickCheck hat gerade für jede der drei Eigenschaften 100 Zufallseingaben generiert und geprüft, dass die angegeben Funktionen auf ihnen `True` liefern.

(Durch diese drei Eigenschaften ist `(++)` sogar schon eindeutig festgelegt. Wir haben also eine *vollständige* Menge von Eigenschaften gefunden.)

Aufgabe 10. *Korrektheit von* `sort`

Schreibe QuickCheck-Tests für `Data.List.sort`! Prüfe dabei folgendes:

- Das Ergebnis ist sortiert.
- Es gehen keine Elemente der Eingabeliste verloren.
- Es werden auch nicht Elemente dupliziert.

Aufgabe 11. *Korrektheit von* `fibs`

Wir haben auf dem ersten Übungsblatt in Aufgabe 8 die Liste `fibs :: [Int]` von Fibonacci-Zahlen implementiert. Wir haben außerdem die Funktion `fib :: Int -> Int` naiv über die Rekursionsgleichung `fib (n+2) = fib (n+1) + fib n` definiert.

Benutze QuickCheck, um zu testen, dass beide Definitionen äquivalent sind, d. h. dass für alle natürlichen Zahlen n das n-te Listenelement von `fibs` (also `fibs !! n`) gleich `fib n` ist.

Tipp. Du musst als Vorbedingung fordern, dass $n \geq 0$ ist. Dazu kannst du entweder die Vorbedingung deiner Eigenschaft voranstellen mit `(n >= 0) ==> ...` (dabei ist `==>` eine Funktion aus `Test.QuickCheck`). Das Problem daran ist, dass von den 100 zufällig generierten Eingaben etliche verworfen werden, weil sie die Vorbedingung nicht erfüllen. Dadurch wird die Eigenschaft für weniger Werte getestet. Besser ist es, gleich nur solche Zufallseingaben zu generieren, die die Vorbedingung erfüllen. In QuickCheck gibt es dafür den Datentyp `NonNegative Int`. Von QuickCheck generierte Zufallswerte von diesem Typ haben die Form `NonNegative n`, wobei `n :: Int` nichtnegativ (Überraschung!) ist. Du kannst dies ausnutzen, wenn du deine Eigenschaft so umschreibst, dass sie den Typ `fib_correct :: NonNegative Int -> Bool` hat.

Außerdem solltest du dich auf Eingaben einer bestimmten Maximalgröße beschränken. Ansonsten könnte die Sonne zu einem roten Riesen werden, bevor QuickCheck mit dem Überprüfen fertig ist.

Bemerkung. In der letzten Übungsaufgabe haben wir ein häufig verwendetes Muster bei der Verwendung von QuickCheck gesehen. Man implementiert eine Funktion zweimal, einmal so, dass die Implementierung offensichtlicherweise korrekt ist, und ein zweites Mal so, dass sich die Funktion schnell ausführen lässt. Dann verwendet man QuickCheck, um zu überprüfen, dass die beiden Implementierungen auf allen (getesteten) Eingaben übereinstimmen.

Aufgabe 12. *Korrektheit der Run-Length-Codierung*

Überprüfe, dass die in Aufgabe 12 von Blatt 1 implementierten Funktionen `encode` und `decode` tatsächlich Inverse voneinander sind!

Tipp. Mit `forAll genRLE :: ([(Int, Char)] -> Bool) -> Property` kannst du eine Eigenschaft nur für alle Werte vom Typ `[(Int, Char)]` testen, die Run-Length-Encodings repräsentieren (d. h. die zweite Komponente jedes Tupels ist eine Zahl ≥ 1). Dabei beschreibt `genRLE = listOf ((,) <$> (getPositive <$> arbitrary) <*> arbitrary)` einen Zufallsgenerator von validen Run-Length-Encodings.

Wir wollen nun Aufgabe 20 von Blatt 1 testen. Dort haben wir den Datentyp

```
data Tree = Nil | Fork Int Tree Tree
```

definiert. Wir müssen zunächst QuickCheck beibringen, wie zufällige Werte vom Typ `Tree` generiert werden können. Dazu müssen wir folgende Typklasse instanziieren

```
class Arbitrary a where
  arbitrary :: Gen a
```

Dabei ist `Gen a` der Typ der zufälligen Generatoren von Werten vom Typ `a`. Folgende Instanzen sind vordefiniert:

```
instance Arbitrary Bool where
  arbitrary = elements [False, True]
  -- ‘elements‘ wählt zufällig ein Element aus der Liste aus

instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b) where
  arbitrary = do
    a <- arbitrary
    b <- arbitrary
    return (a, b)
  -- alternativ: arbitrary = (,) <$> arbitrary <*> arbitrary
```

```haskell
instance (Arbitrary a, Arbitrary b) => Arbitrary (Either a b) where
  arbitrary = oneof [Left <$> arbitrary, Right <$> arbitrary]
  -- ‘oneof‘ wählt zwischen zwei Möglichkeiten: entweder wir generieren ein
  -- zufälliges Element ‘x :: a‘ und geben dann ‘Left x :: Either a b‘ zurück,
  -- oder wir generieren ‘y :: b‘ und geben ‘Right y :: Either a b‘ zurück.
```

Aufgabe 13. *QuickCheck mit binären Bäumen*

a) Verstehe folgende `Arbitrary`-Instanz:

```haskell
instance Arbitrary Tree where
  arbitrary = oneof [genNil, genFork]
    where
      genNil = return Nil
      genFork = do
        m <- arbitrary
        l <- arbitrary
        r <- arbitrary
        return (Fork m l r)
```

b) Teste folgende Eigenschaft: `numberOfLeaves l + numberOfLeaves r = numberOfLeaves (Fork m l r)` für alle `l, r :: Tree` und `m :: Int`.

c) Formuliere eine Eigenschaft, die die Höchsttiefe-Funktion aus Aufgabe 20b) von Blatt 1 besitzen sollte, und überprüfe sie mit QuickCheck.

Aufgabe 14. *Funktoraxiome*

Zeige, dass deine Implementierung von `tmap :: (a -> b) -> Tree  a -> Tree b` aus Aufgabe 21 die Funktoraxiome erfüllt, d. h.

1. `tmap id = id`
2. `tmap g . tmap f = tmap (g . f)` für alle `f :: a -> b` und `g :: b -> c`

Wegen einer magischen Eigenschaft des Haskell-Typsystems musst du nur Axiom 1 mit QuickCheck nachprüfen. Axiom 2 ist dann vollautomatisch erfüllt.

Hinweis. Du musst zunächst die obige `Arbitrary`-Instanz von `Tree` wie folgt abändern:

```haskell
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = ... -- Rest wie oben
```

Aufgabe 15. *Korrektheit von* `partition`

Die Funktion `partition :: (a -> Bool) -> [a] -> ([a], [a])` aus `Data.List` teilt eine Liste auf in diejenigen Elemente, die ein gegebenes *Prädikat* `p :: a -> Bool` erfüllen, und diejenigen, die es nicht erfüllen. Überprüfe:

a) Sei `(as, bs) = partition p xs`. Die Elemente in `as` erfüllen alle `p`, die Elemente in `bs` erfüllen `p` nicht.

b) Es geht kein Element durch `partition` „verloren“.

Es genügt, die Aussagen mit `Int` für `a` und `p m = m >= 0` zu überprüfen.

Aufgabe 16. *Algebraische Axiome*

Übersetze die Monoid-Axiome in Haskell-Code! Du findest die `Monoid`-Typklasse in Aufgabe 27 auf Blatt 1. Frag nach, falls du nicht weißt, was ein Monoid ist!

```haskell
prop_assoc :: (Monoid m, Eq m) => m -> m -> m -> Bool
prop_assoc = ...

prop_mempty_neutral_left :: (Monoid m, Eq m) => m -> Bool
prop_mempty_neutral_left = ...

prop_mempty_neutral_right :: (Monoid m, Eq m) => m -> Bool
prop_mempty_neutral_right = ...
```

Überprüfe, dass `[Char]` ein Monoid ist.

Hinweis. Verwende `quickCheck (prop_assoc :: [Char] -> [Char] -> [Char] -> Bool)`. Mit der expliziten Typsignatur legst du die Typvariable `m` fest, d. h. du teilst mit, welche Monoid-Instanz du überprüfen willst.

Aufgabe 17. *QuickCheck all the things!*

Überprüfe alle von dir bearbeiteten Aufgaben von Blatt 1 auf ihre Korrektheit.

Hinweis. Im Besonderen eignen sich die Aufgaben 39 und 46.

3 2D-Spiele mit Gloss

Aufgabe 18. *Flappy Bird, Tetris, Snake, Asteroids*

Implementiere ein 2D-Spiel deiner Wahl. Passe dazu folgende Vorlage an, die du auch auf `https://github.com/curry-club-aux/haskell-workshop/blob/gh-pages/gloss-game/game.hs` findest:

```haskell
module Main where

import Graphics.Gloss.Interface.Pure.Game

main :: IO ()
main = play displayMode white fps initialState render handleInput step
  where width = 800
        height = 600
        displayMode = InWindow "Game" windowSize windowPos
        windowSize = (width, height)
        windowPos = (10, 10)
        fps = 100

-- Speichere in dieser Datenstruktur den Zustand deines Spiels, also Positionen
-- und Geschwindigkeit von deiner Spielfigur, den Gegnern, von sich bewegenden
-- Plattformen, die Anzahl an verbleibenden <3 usw.
data GameState = GameState
  { squarePosition :: Point -- ^ Point ist ein Alias für (Float, Float)
  } deriving (Show)

-- Der Zustand zu Beginn des Spiels.
initialState :: GameState
initialState = GameState
  { squarePosition = (0, 0)
  }

-- Gloss rendert Pictures, wenn man nichts tut, bei (0, 0) also in der Mitte des
-- Bildschirms, darum müssen wir mit translate x y bild unser das Bild an die
-- richtige Stelle verschieben. Unter
-- https://hackage.haskell.org/package/gloss-1.9.4.1/docs/
-- Graphics-Gloss-Data-Picture.html findest du eine Dokumentation aller
-- Funktionen, mit denen du ein Picture produzieren kannst. makeColor nimmt den
-- Rot-, Grün-, Blau- und Transparenz-Anteil einer Farbe als Floats zwischen 0
-- und 1.0.
render :: GameState -> Picture
render (GameState (x, y)) =
  translate x y $ color (makeColor 0.5 1.0 0.1 1.0) (rectangleSolid 100 40)
```

```haskell
-- In dieser Funktion beschreibst du, wie sich der Zustand ändern soll, wenn
-- der Benutzer die Maus bewegt, klickt oder eine Taste drückt.
-- Unter https://hackage.haskell.org/package/gloss-1.9.4.1/docs/
-- Graphics-Gloss-Interface-Pure-Game.html#t:Event
-- findest du die Dokumentation zu Event.
handleInput :: Event -> GameState -> GameState
handleInput (EventKey (SpecialKey k) _ _ _) (GameState (x, y))
  = case k of
      KeyUp    -> GameState (x, y + 5)
      KeyDown  -> GameState (x, y - 5)
      KeyLeft  -> GameState (x - 5, y)
      KeyRight -> GameState (x + 5, y)
      _        -> GameState (x,y)
handleInput _ gs = gs

-- Beschreibe, wie sich der Zustand ändert, wenn Zeit verstreicht.
-- 'step' bekommt im ersten Argument die Änderung der Zeit,
-- seit dem das letzte Mal step aufgerufen wurde in Sekunden.
step :: Float -> GameState -> GameState
step _ gs = gs -- keine Zustandsänderung
```

4 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.

5 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.

5.1 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 Betriebssystemthread. 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 Betriebssystemthreads verwenden
$ ./Main +RTS -N -RTS    # sinnvolle Anzahl Betriebssystemthreads verwenden

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

Aufgabe 19. *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 20. *Paralleles Map*

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

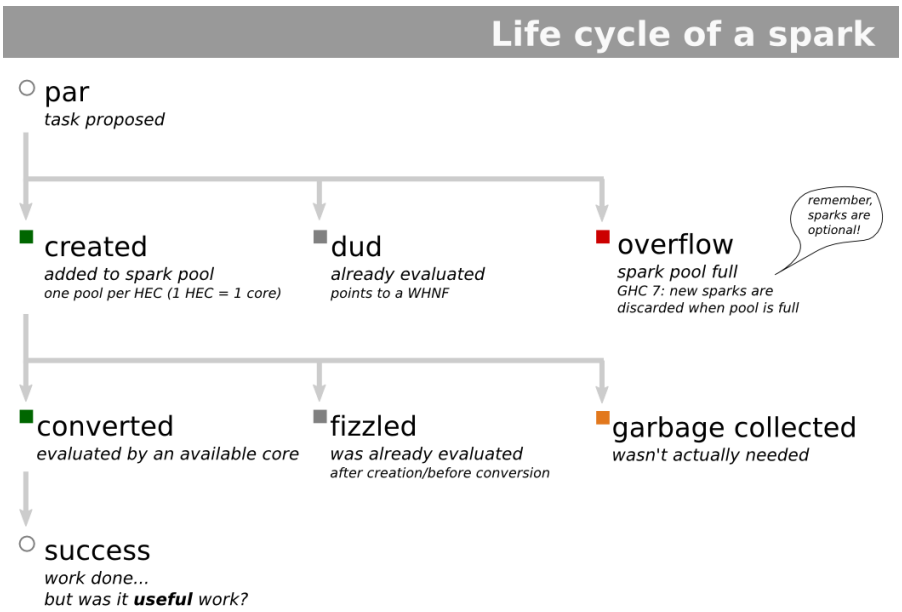
Auf einem Mehrkerncomputer sollte also

```
> 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`

5.2 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 Betriebssystemlevel verteilt.

In speziellen Anwendungsfällen ist es nötig, Betriebssystemthreads 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 Betriebssystemthread. 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 das Laufzeitsystem mit `+RTS -N -RTS` anweisen, mehrere Betriebssystemthreads zu verwenden.

Aufgabe 21. *Hallo Welt aus zwei Threads*

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

Aufgabe 22. *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 23. *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 entweder leer sein oder einen Wert enthalten. Man erstellt sie mit `newEmptyMVar :: IO (MVar a)` oder, wenn man die Variable gleich initialisieren 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 24. *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 25. *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 26. *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 27. *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 28. *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 29. *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 Zuhörerinnen 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
```

5.3 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.