

Reaktive Programmierung
Vorlesung 2 vom 16.04.15: Monaden und Nebenläufigkeit in Haskell

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- ▶ Teil I: Grundlegende Konzepte
 - ▶ Was ist Reaktive Programmierung?
 - ▶ Nebenläufigkeit und Monaden in Haskell
 - ▶ Funktional-Reaktive Programmierung
 - ▶ Einführung in Scala
 - ▶ Die Scala Collections
 - ▶ ScalaTest und ScalaCheck
- ▶ Teil II: Nebenläufigkeit
- ▶ Teil III: Fortgeschrittene Konzepte

Speisekarte

- ▶ Das Geheimnis der Monade
- ▶ Concurrent Haskell

Zustandsübergangsmo- naden

- ▶ Aktionen ($IO\ a$) sind keine schwarze Magie.
- ▶ Grundprinzip: Systemzustand Σ wird explizit behandelt.

$$f :: a \rightarrow IO\ b \cong f :: (a, \Sigma) \rightarrow (b, \Sigma)$$

Folgende **Invarianten** müssen gelten:

- ▶ Systemzustand darf **nie dupliziert** oder **vergessen** werden.
- ▶ Auswertungsreihenfolge muss erhalten bleiben.
- ▶ Komposition muss **Invarianten** erhalten
 \rightsquigarrow **Zustandsübergangsmo-
naden**

Komposition von Zustandsübergängen

- ▶ Im Prinzip Vorwärtskomposition:

$$\begin{aligned} (\gg) &:: ST\ s\ a \rightarrow (a \rightarrow ST\ s\ b) \rightarrow ST\ s\ b \\ (\gg) &:: (s \rightarrow (a, s)) \rightarrow (a \rightarrow s \rightarrow (b, s)) \rightarrow (s \rightarrow (b, s)) \\ (\gg) &:: (s \rightarrow (a, s)) \rightarrow ((a, s) \rightarrow (b, s)) \rightarrow (s \rightarrow (b, s)) \end{aligned}$$

- ▶ Damit $f \gg g = \text{uncurry } g \circ f$
- ▶ Aber: ST kann kein Typsynonym sein
- ▶ Nötig: **abstrakter Datentyp** um Invarianten zu erhalten

ST als Abstrakter Datentyp

- ▶ Datentyp verkapseln:

```
newtype ST s a = ST (s -> (a, s))
```

- ▶ Hilfsfunktion (Selektor)

```
unwrap :: ST s a -> (s -> (a, s))  
unwrap (ST f) = f
```

- ▶ Damit ergibt sich

```
f >> g = ST (uncurry (unwrap . g) o unwrap f)  
return a = ST (\s -> (a, s))
```

Aktionen

- ▶ Aktionen: Zustandstransformationen auf der Welt
- ▶ Typ `RealWorld#` repräsentiert Außenwelt
 - ▶ Typ hat genau einen Wert `realworld#`, der nur für initialen Aufruf erzeugt wird.
 - ▶ Aktionen: `type IO a = ST RealWorld# a`
- ▶ Optimierungen:
 - ▶ ST `s a` durch **in-place-update** implementieren.
 - ▶ IO-Aktionen durch **einfachen Aufruf** ersetzen.
 - ▶ Compiler darf keine Redexe duplizieren!
 - ▶ Typ IO stellt **lediglich** Reihenfolge sicher.

Was ist eigentlich eine Monade?

- ▶ ST modelliert **imperative Konzepte**.
- ▶ **Beobachtung**: Andere Konzepte können **ähnlich** modelliert werden:
 - ▶ **Ausnahmen**: $f :: a \rightarrow \text{Maybe } b$ mit Komposition

```
(\gg) :: Maybe a -> (a -> Maybe b) -> Maybe b  
Just a >> f = f a  
Nothing >> f = Nothing
```

Monads: The Inside Story

```
class Monad m where
  (≫) :: m a → (a → m b) → m b
  return :: a → m a
  (≫) :: m a → m b → m b
  fail :: String → m a

  p ≫ q = p ≫ λ_ → q
  fail s = error s
```

Folgende Gleichungen müssen (sollten) gelten:

```
return a ≫ k = k a
m ≫ return = m
m ≫ (λx → k x ≫ h) = (m ≫ k) ≫ h
```

9 [32]

Beispiel: Speicher und Referenzen

► Signatur:

```
type Mem a
instance Monad
```

► Referenzen sind abstrakt:

```
type Ref
newRef :: Mem Ref
```

► Speicher liest/schreibt String:

```
readRef :: Ref → Mem String
writeRef :: Ref → String → Mem ()
```

10 [32]

Implementation der Referenzen

Speicher: Liste von Strings, Referenzen: Index in Liste.

```
type Mem = ST [String] — Zustand
type Ref = Int
```

```
newRef = ST (λs → (length s, s+[""]))
readRef r = ST (λs → (s !! r, s))
writeRef r v = ST (λs → ((),
    take r s ++ [v] ++ drop (r+1) s))
run :: Mem a → a
run (ST f) = fst (f [])
```

11 [32]

IORef — Referenzen

► Datentyp der Standardbibliothek (GHC)

```
import Data.IORef
```

```
data IORef a
```

```
newIORef :: a → IO (IORef a)
readIORef :: IORef a → IO a
writeIORef :: IORef a → a → IO ()

modifyIORef :: IORef a → (a → a) → IO ()
atomicModifyIORef :: IORef a → (a → (a, b)) → IO b
```

► Implementation: "echte" Referenzen.

12 [32]

Beispiel: Referenzen

```
fac :: Int → IO Int
fac x = do acc ← newIORef 1
    loop acc x where
        loop acc 0 = readIORef acc
        loop acc n = do t ← readIORef acc
            writeIORef acc (t * n)
            loop acc (n-1)
```

13 [32]

Die Identitätsmonade

► Die allereinfachste Monade:

```
type Id a = a
```

```
instance Monad Id where
    return a = a
    b ≫ f = f b
```

14 [32]

Die Listenmonade

► Listen sind Monaden:

```
instance Monad [] where
    m ≫ f = concatMap f m
    return x = [x]
    fail s = []
```

► Intuition: $f :: a \rightarrow [b]$ Liste der möglichen Resultate

► Reihenfolge der Möglichkeiten relevant?

15 [32]

Fehlermonaden

► Erste Näherung: Maybe

► Maybe kennt nur Nothing, daher strukturierte Fehler:

```
data Either a b = Left a | Right b
type Error a = Either String a
```

```
instance Monad (Either String) where
    (Right a) ≫ f = f a
    (Left l) ≫ f = Left l
    return b = Right b
```

► Nachteil: Fester Fehlertyp

► Lösung: Typklassen

16 [32]

Exkurs: Was *genau* ist eigentlich eine Monade?

- ▶ Monade: Konstrukt aus Kategorientheorie
- ▶ Monade \cong (verallgemeinerter) Monoid
- ▶ Monade: gegeben durch **algebraische Theorien**
 - ▶ Operationen endlicher (beschränkter) Arität
 - ▶ Gleichungen
- ▶ Beispiele: Maybe, List, Set, State, ...
- ▶ Monaden in Haskell: computational monads
 - ▶ Strukturierte Notation für Berechnungsparadigmen
 - ▶ Beispiel: Rechner mit Fehler, Nichtdeterminismus, Zustand, ...

17 [32]

Konzepte der Nebenläufigkeit

- ▶ **Thread (lightweight process)** vs. **Prozess**

Programmiersprache/Betriebssystem (z.B. Java, Haskell, Linux)	Betriebssystem
gemeinsamer Speicher	getrennter Speicher
Erzeugung billig	Erzeugung teuer
mehrere pro Programm	einer pro Programm
- ▶ Multitasking:
 - ▶ **präemptiv**: Kontextwechsel wird erzwungen
 - ▶ **kooperativ**: Kontextwechsel nur freiwillig

18 [32]

Zur Erinnerung: **Threads in Java**

- ▶ Erweiterung der Klassen Thread oder Runnable
- ▶ Gestartet wird Methode run() — durch eigene überladen
- ▶ Starten des Threads durch Aufruf der Methode start()
- ▶ Kontextwechsel mit yield()
- ▶ Je nach JVM kooperativ **oder** präemptiv.
- ▶ Synchronisation mit synchronize

19 [32]

Threads in Haskell: Concurrent Haskell

- ▶ **Sequentielles Haskell**: Reduktion eines Ausdrucks
 - ▶ Auswertung
- ▶ **Nebenläufiges Haskell**: Reduktion eines Ausdrucks an **mehreren Stellen**
- ▶ ghc implementiert Haskell-Threads
- ▶ Modul Control.Concurrent enthält Basisfunktionen
- ▶ Wenige Basisprimitive, darauf aufbauend Abstraktionen

20 [32]

Wesentliche Typen und Funktionen

- ▶ Jeder Thread hat einen Identifier: abstrakter Typ ThreadId
- ▶ Neuen Thread erzeugen: forkIO :: IO() → IO ThreadId
- ▶ Thread stoppen: killThread :: ThreadId → IO ()
- ▶ Kontextwechsel: yield :: IO ()
- ▶ Eigener Thread: myThreadId :: IO ThreadId
- ▶ Warten: threadDelay :: Int → IO ()

21 [32]

Rahmenbedingungen

- ▶ **Zeitscheiben**:
 - ▶ Tick: Default 20ms
 - ▶ Contextswitch pro Tick bei Heapallokation
 - ▶ Änderungen per Kommandozeilenoptionen: +RTS -V<time> -C<time>
- ▶ **Blockierung**:
 - ▶ Systemaufrufe blockieren alle Threads
 - ▶ Mit threaded library (-threaded) nicht alle
 - ▶ **Aber**: Haskell Standard-IO blockiert **nur den aufrufenden Thread**

22 [32]

Concurrent Haskell — erste Schritte

- ▶ Ein einfaches Beispiel:

```
write :: Char → IO ()
write c = putChar c >> write c

main :: IO ()
main = forkIO (write 'X') >> write 'O'
```

- ▶ Ausgabe ghc: (X*|O*)*

23 [32]

Synchronisation mit MVars

- ▶ **Basissynchronisationsmechanismus** in Concurrent Haskell
 - ▶ Alles andere abgeleitet
- ▶ MVar a veränderbare Variable (vgl. IOREf a)
- ▶ Entweder **leer** oder **gefüllt** mit Wert vom Typ a
- ▶ Verhalten beim Lesen und Schreiben

Zustand vorher:	leer	gefüllt
Lesen	blockiert (bis gefüllt)	danach leer
Schreiben	danach gefüllt	blockiert (bis leer)
- ▶ NB. Aufwecken blockierter Prozesse einzeln in FIFO

24 [32]

Basisfunktionen MVars

- ▶ Neue Variable erzeugen (leer oder gefüllt):

```
newEmptyMVar :: IO (MVar a)
newMVar      :: a → IO (MVar a)
```

- ▶ Lesen:

```
takeMVar :: MVar a → IO a
```

- ▶ Schreiben:

```
putMVar :: MVar a → a → IO ()
```

25 [32]

Abgeleitete Funktionen MVars

- ▶ Nicht-blockierendes Lesen/Schreiben:

```
tryTakeMVar :: MVar a → IO (Maybe a)
tryPutMVar  :: MVar a → a → IO Bool
```

- ▶ Änderung der MVar:

```
swapMVar    :: MVar a → a → IO a
withMVar    :: MVar a → (a → IO b) → IO b
modifyMVar  :: MVar a → (a → IO (a, b)) → IO b
```

- ▶ **Achtung:** race conditions

26 [32]

Ein einfaches Beispiel ohne Synchronisation

- ▶ Nebenläufige Eingabe von der Tastatur

```
echo :: String → IO ()
echo p = forever (do
  putStrLn ("**_Please_enter_line_#_#p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_n (putStr (p+ ":#_# line+_#_"))

main :: IO ()
main = forkIO (echo "2") >> echo "1"
```

- ▶ **Problem:** gleichzeitige Eingabe
- ▶ **Lösung:** MVar synchronisiert Eingabe

27 [32]

Ein einfaches Beispiel mit Synchronisation

- ▶ MVar voll ⇔ Eingabe möglich
- ▶ Also: initial voll
- ▶ Inhalt der MVar irrelevant: MVar ()

```
echo :: MVar () → String → IO ()
echo flag p = forever (do
  takeMVar flag
  putStrLn ("**_Please_enter_line_#_# p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_n (putStr (p+ ":#_# line+_#_"))
  putMVar flag ())

main :: IO ()
main = do flag ← newMVar ()
  forkIO (echo flag "3") >> forkIO (echo flag "2") >>
  echo flag "1"
```

28 [32]

Das Standardbeispiel

- ▶ Speisende Philosophen
- ▶ Philosoph *i*:
 - ▶ vor dem Essen *i*-tes und (*i* + 1) mod *n*-tes Stäbchen nehmen
 - ▶ nach dem Essen wieder zurücklegen
- ▶ Stäbchen modelliert als MVar ()

29 [32]

Speisende Philosophen

```
philosopher :: [MVar ()] → Int → IO ()
philosopher chopsticks i = forever (do
  let num_phil = length (chopsticks)
  — Thinking:
  putStrLn ("Phil_#_# show i +_#_# thinks.")
  randomRIO (10, 200) >>= threadDelay
  — Get ready to eat:
  takeMVar (chopsticks !! i)
  takeMVar (chopsticks !! ((i+1) 'mod' num_phil))
  — Eat:
  putStrLn ("Phil_#_# show i +_#_# eats..")
  randomRIO (10, 200) >>= threadDelay
  — Done eating:
  putMVar (chopsticks !! i) ()
  putMVar (chopsticks !! ((i+1) 'mod' num_phil)) ())
```

30 [32]

Speisende Philosophen

- ▶ Hauptfunktion: *n* Stäbchen erzeugen
 - ▶ Anzahl Philosophen in der Kommandozeile
- ```
main = do
 a:_ ← getArgs
 let num = read a
 chopsticks ← replicateM num (newMVar ())
 mapM_ (forkIO ∘ (philosopher chopsticks)) [0.. num-1]
 block
```
- ▶ Hilfsfunktion `block`: blockiert aufrufenden Thread
- ```
block :: IO ()
block = newEmptyMVar >>= takeMVar
```
- ▶ NB: Hauptthread terminiert — Programm terminiert!

31 [32]

Zusammenfassung

- ▶ Monaden und andere Kuriositäten
 - ▶ Zustandsmonade - Referenzen
 - ▶ Fehlermonaden
- ▶ **Concurrent Haskell** bietet
 - ▶ **Threads** auf Quellsprachenebene
 - ▶ Synchronisierung mit MVars
 - ▶ Durch **schlankes Design** einfache Implementierung
- ▶ Funktionales Paradigma erlaubt **Abstraktionen**
 - ▶ Beispiel: Semaphoren
- ▶ Nächste Woche: Funktional-Reaktive Programmierung.

32 [32]