Reaktive Programmierung
Vorlesung 2 vom 16.04.15: Monaden und Nebenlaeufigkeit 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

Zustandsübergangsmonaden

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

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

  Folgende Invarianten müssen gelten:

  - Systemzustand darf nie dupliziert oder vergessen werden.
  - Auswertungsreihenfolge muss erhalten bleiben.

- Komposition muss Invarianten erhalten
  $\rightsquigarrow$ Zustandsübergangsmonaden

Komposition von Zustandsübergängen

- Im Prinzip Vorwärtskomposition:

```
(>>=) :: ST s a→ (a→ ST s b)→ ST s b
(>>=) :: (s→ (a, s))→ (a→ s→ (b, s))→ (s→ (b, s))
(>>=) :: (s→ (a, s))→ ((a, s)→ (b, s))→ (s→ (b, s))
```

- Damit f >>=g =uncurry g ∘ 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) ∘ unwrap f)
return a = ST (λs → (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-> Maybe b` mit Komposition

```
(>>=) :: 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:

$$\begin{aligned} \text{return } a \gg\!= k &= k\ a \\ m \gg\!= \text{return} &= m \\ m \gg\!= (\lambda x \to k\ x \gg\!= h) &= (m \gg\!= k) \gg\!= h \end{aligned}$$

Beispiel: Speicher und Referenzen

- Signatur:

```
type Mem a
instance Mem Monad
```

- Referenzen sind abstrakt:

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

- Speicher liest/schreibt String:

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

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 [])
```

`IORef` — Referenzen

- Datentyp der Standardbücherei (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.

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)
```

Die Identitätsmonade

- Die allereinfachste Monade:

```
type Id a = a

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

Die Listenmonade

- Listen sind Monaden:

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

- Intuition: f :: a → [b] Liste der möglichen Resultate

- Reihenfolge der Möglichkeiten relevant?

Fehlermonaden

- Erste Nährung: 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

Exkurs: Was *genau* ist eigentliche eine Monade?

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

Konzepte der Nebenläufigkeit

- Thread (lightweight process) vs. Prozess

Thread (lightweight process)	vs.	Prozess
Programmiersprache/Betriebssystem		Betriebssystem
(z.B. Java, Haskell, Linux)		
gemeinsamer Speicher		getrennter Speicher
Erzeugung billig		Erzeugung teuer
mehrere pro Programm		einer pro Programm

- Multitasking:
  - präemptiv: Kontextwechsel wird erzwungen
  - kooperativ: Kontextwechsel nur freiwillig

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`

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

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 ()

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

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`: $(\mathtt{X}^*|\mathtt{O}^*)^*$

Synchronisation mit MVars

- Basissynchronisationmechanismus 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

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 ()
```

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

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
echo :: String→ IO ()
echo p = forever (do
  putStrLn ("***␣Please␣enter␣line␣for␣"++p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n (putStr (p++ ":"++ line++"␣")))

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

- Problem: gleichzeitige Eingabe

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
echo :: String→ IO ()
echo p = forever (do
  putStrLn ("***␣Please␣enter␣line␣for␣"++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

Ein einfaches Beispiel mit Synchronisation

- `MVar` voll $\Leftrightarrow$ 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"
```

Das Standardbeispiel

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

Speisende Philosophen

```
philo :: [MVar ()] → Int→ IO ()
philo 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)) ())
```

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 ∘ (philo chopsticks)) [0.. num−1]
  block
```

- Hilfsfunktion `block`: blockiert aufrufenden Thread

```
block :: IO ()
block = newEmptyMVar ≫= takeMVar
```

- NB: Hauptthread terminiert — Programm terminiert!

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.