

Fortgeschrittene Techniken der Funktionalen Programmierung

Vorlesung vom 27.10.09:
Monads — The Inside Story

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

1

Heute in diesem Theater

- ▶ Die Geheimnisse der Monaden
- ▶ Endlich Zuweisungen
- ▶ Flucht aus Alcatraz – IO für Erwachsene

2

Fahrplan

- ▶ **Teil I: Monaden und fortgeschrittene Typen**
 - ▶ Einführung, Wiederholung
 - ▶ **Zustände, Zustandsübergänge und IO**
 - ▶ Reader/Writer, Nichtdeterminismus, Ausnahmen
 - ▶ Monadentransformer
- ▶ Teil II: Fortgeschrittene Datenstrukturen
- ▶ Teil III: Nebenläufigkeit
- ▶ Teil IV: The Future of Programming

3

Zustandsübergangsmonaden

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

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

Folgende **Invarianten** müssen gelten:

- ▶ Systemzustand darf **nie dupliziert** oder **vergessen** werden.
- ▶ Auswertungsreihenfolge muß erhalten bleiben.
- ▶ **Komposition** muss **Invarianten** erhalten
~~> **Zustandsübergangsmonaden**

4

Zustandsübergangsmonaden

- ▶ Typ:

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

$$a \rightarrow \text{ST } s \ b = a \rightarrow s \rightarrow (b, s) \cong (a, s) \rightarrow (b, s)$$

Parametrisiert über Zustand s und Berechnungswert a.

- ▶ Komposition durch

```
(>=>) :: ST s a -> (a -> ST s b) -> ST s b
```

5

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

6

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 (\s -> (a, s))
```

7

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.

8

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

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

- Benötigen Typklassen für Typkonstruktoren...

9

Konstruktorklassen

- Konstruktorklassen: Typklassen für Typkonstruktoren (kinds)

- Beispiel:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)

instance Functor [ ] where — Kein 'echtes' Haskell!
  fmap f [] = []
  fmap f (x:xs) = f x : map f xs
```

- Erweiterung des Typsystems (bleibt entscheidbar)

- Für Zustandstransformer ST:

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

10

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 >>= \_ -> q
fail s = error s
```

Folgende Gleichungen müssen (sollten) gelten:

```
return a >>= k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

11

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 ()
```

12

Implementation der Referenzen

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

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

newRef = ST (\s -> (length s, s+[""]))
readRef r = ST (\s -> (s !! r, s))
writeRef r v = ST (\s -> ((,
    take r s ++ [v] ++ drop (r+1) s))

run :: Mem a -> a
run (ST f) = fst (f [])
```

13

IORef — Referenzen

- Datentyp der Standardbibliothek (GHC, Hugs)

```
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.

14

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)
```

15

Flucht aus Alcatraz

- Aus dem IO-Monaden gibt es keinen Ausweg.

- Im Gegensatz zu z.B. Maybe:

```
fromMaybe :: a -> Maybe a -> a
```

- Das ist manchmal unpraktisch: Initialisierungen etc.

- Für ST gibt es

```
fixST :: (a -> ST s a) -> ST s a — Fixpunkt
runST :: (forall s . ST s a) -> a — NB: Typ!
```

- Für IO gibt es...

16

Unsichere Aktionen

- ▶ Signatur:

```
import System.IO.Unsafe(unsafePerformIO)
```

```
unsafePerformIO :: IO a → a
```

- ▶ **Warnung:** gefährlich und nicht typsicher!

```
test :: IORef [a]
test = unsafePerformIO $ newIORef []

main = do writeIORef test [42]
         bang ← readIORef test
         putStrLn (bang :: [Char])
```

17

Verwendung von unsafePerformIO

- ▶ IO-Aktionen, die nur
 - ▶ einmal durchgeführt werden sollen, und
 - ▶ von anderen IO-Aktionen unabhängig sind
 - ▶ Beispiel: Konfigurationsdatei lesen
- ▶ Allokierung globaler Ressourcen (z.B. Referenzen).
- ▶ Debugging (traces, logfiles).
- ▶ Enjoy responsibly!

18

Benutzung von unsafePerformIO

- ▶ Allokierung globaler Referenzen:

```
-- -- Generate a new identifier.
newId :: IO Int
newId = atomicModifyIORef r $ \i → (i+1, i) where
    r = unsafePerformIO $ newIORef 1
{-# NOINLINE newId #-}
```

- ▶ **NOINLINE** beachten — Optimierungen verhindern.

- ▶ Debugging:

```
trace :: String → a → a
trace s x = unsafePerformIO $ putStrLn s >> return x
```

- ▶ Schon vordefiniert (Debug.Trace).

19

Zusammenfassung & Ausblick

- ▶ Blick hinter die Kulissen von IO
- ▶ Monaden und andere Kuriositäten
- ▶ Referenzen
- ▶ unsafePerformIO
- ▶ Nächstes Mal: Mehr Monaden...

20