Exklusiv:
Monads — The Inside Story

Tagesprogramm

Heute in diesem Theater:

- Die Geheimnisse der Monaden
- Endlich Zuweisung
- Flucht aus Alcatraz – `IO` für Erwachsene

Zustandsübergangsmonaden

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

  `f :: a-> IO b` $\cong$ `f :: (a,` Σ`) -> (b,` Σ`)`

  Folgende Invarianten müssen gelten:

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

- Komposition muss Invarianten erhalten.

  $\rightsquigarrow$ Zustandsübergangsmonaden

Zustandsübergangsmonaden

- Typ:

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

$$\texttt{a -> ST s b} \cong \texttt{a-> s-> (b, s)} \cong \texttt{(a, s)-> (b, s)}$$

Parametrisiert über Zustand `s` und Berechnungswert `a`.

- Komposition durch

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

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

  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

```
instance Monad (ST s) where
  f >>= g = ST (uncurry (unwrap. g) . 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 ein Monad?

`ST` modelliert imperative Konzepte.

Beobachtung: Andere Konzepte können ähnlich modelliert werden:

- Nichtdeterminismus: $f : a \Rightarrow \mathbb{P}(b)$ mit Komposition

  $f : \mathbb{P}(a) \Rightarrow (a \Rightarrow \mathbb{P}(b)) \Rightarrow (\mathbb{P}(b))$

  $A \texttt{>>=} f = \{f\,a \mid a \in A\}$

- Ausnahmen: `f :: a-> Maybe b` mit Komposition

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

- Benötigen Typklassen für Typkonstruktoren. . .

Konstruktorklassen

- Monaden sind Konstruktorklassen:
  Typklassen für Typkonstruktoren (kinds)
- Beispiel:

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

instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = f x : map f xs
```

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
```

Beispiel: Referenzen

Speicher und Referenzen.

Signatur:

```
type Mem
instance Mem Monad

newRef   :: Mem Ref
readRef  :: Ref -> Mem String
writeRef :: Ref -> String-> Mem ()
```

Implementation der Referenzen

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

```
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 [])
```

IORef — Referenzen

Datentyp der Standardbücherei (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.

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)
```

Zeigen.

Ausbruch 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 ...

Unsichere Aktionen

```
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])
```

Verwendung von `unsafePerformIO`

Nur verwenden bei:

- Einmalig durchgeführter, von anderen IO-Aktionen unabhängiger Aktion (zum Beispiel Konfigurationsdatei lesen).
- `IO`-Aktionen, die nur einmal durchgeführt werden sollen.
- Alloziierung globaler Ressourcen (z.B. Referenzen).
- Debugging (traces, logfiles).

Verwendung von `unsafePerformIO`

Alloziierung globaler Referenzen:

```
-- | Generate a new identifier.
newId :: IO Int
newId = do 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)
```

aber auch vordefiniert (`Debug.Trace`).

Zusammenfassung & Ausblick

- Blick hinter die Kulissen von `IO`
- Monaden und andere Kuriositäten
- Referenzen
- `unsafePerformIO`
- Nächstes Mal: Nebenläufigkeit und Monadentransformer ?!?