Kapitel 7: Monaden

Lernziele dieses Kapitels

1. Typen und Basisfunktionen
2. Syntax: `do`, `let`, Blöcke und Gültigkeitsbereiche
3. Referenzielle Transparenz bei Monaden
4. Programmierung von Ein-/Ausgabeoperationen, Synchronisationsprobleme
5. Kombinatoren auf Monaden
6. Definition eigener Monaden
   - Implementierung der Monadenoperationen
   - Beispiele: Fehlermonade, Zustandsmonade
   - Kombination mehrerer Monaden, insbesondere mit der IO-Monade
   - Monadentransformer

Motivation für Monaden

- Bestimmung der Reihenfolge von Berechnungen
  - Sinnvolle Ein-/Ausgabeoperationen trotz Laziness
  - Reassignment bei speziellen, monadischen Arrays (numerisches Rechnen)
  - Explizites Löschen nicht mehr benötigter Datenobjekte
- Verwendung von Zuständen
  - Komfortabler Umgang mit globalen Datenobjekten
  - Separierung von orthogonalen Aspekten (Crosscutting-Concerns), z.B. Protokollierung, Debugging, Ausnahmebehandlung
  - Generierung frischer Namen
  - Graphenalgorithmen: Markierung, Identifizierung von Strukturen
- Ersatz für den unschönen sog. Continuation-Passing-Style

Unterschied zur imperativen Programmierung

- Einbettung nicht-monadischer Funktionen, am Typ erkennbar
- Monadische Berechnungen können funktional komponiert werden
  - Monadenkombinatoren, z.B. `mapM`, `foldM` (Fol. 7.18-21)
  - Monadische Operationen sind selbst manipulierbare Programmwerte
  - Trennung dieser Werte von den Laufzeitwerten
- Mögliche Abhängigkeiten können beschränkt werden:
  eine bestimmte Monade kann nur bestimmte Operationen implementieren
- Lokale Verwendung von Monaden (nicht IO-Monade)
  Initialisierung, Benutzung, Beendung
- Monaden können geschachtelt werden
- Eine Monade kann mit einem speziellen Service / Feature verbunden werden

Basisdefinitionen

- Typklasse `Monad` für Monaden mit Methoden `return` und `(>>=)` (bind)
- Erzeugung einer Monade

  ```
  return :: Monad m => a -> m a
  ```

  - nimmt einen Wert x
  - liefert eine monadische Operation, die nur x als Rückgabewert hat
- Komposition zweier monadischer Berechnungen

  ```
  (>>=) :: Monad m => m a -> (a -> m b) -> m b
  ```

  - zweite Berechnung benutzt Rückgabewert der ersten
  - Rückgabewert ist Rückgabewert der zweiten Berechnung

Die do-Notation

Beispielberechnung:

```
foo a = return a >>= (\b -> (f b >>= (\c -> g b c)))
```

in do-Notation:

```
foo b = do { c <- f b; g b c }
```

im Layout-Stil:

```
foo b = do
          c <- f b
          g b c
```

falls auch der Wert c zurückgegeben werden soll:

```
foo b = do
          c <- f b
          d <- g b c
          return (c,d)
```

Gültigkeitsbereiche von Variablen

jede Zeile in der `do`-Notation schafft einen neuen Gültigkeitsbereich (∗), Bsp.:

```
foo r = do
          x <- f r
          x <- g x
          x <- h x
          return x
```

wegen der Entsprechung zu:

```
foo r = f r >>= ((\x -> g x)
             >>= ((\x -> h x)
             >>= (\x -> return x)))
```

(∗) Ausnahme: innerhalb eines `let`-Blocks ist die Reihenfolge beliebig

Monade speichert Hintergrundinformation

Beispiel mit monadischen Teilfunktionen `searchInMonad` und `updateMonad`

```
foo x = do
        r <- searchInMonad x
        let s = f r
            t = g r
        updateMonad s
        return t
```

- Zugriff auf Hintergrundinformation: `searchInMonad` und `updateMonad`
- `f` und `g` sind Funktionen außerhalb der Monade
- bei Anwendung von `f` und `g` bleibt Hintergrundinformation unverändert
- `let` nicht strikt: Anwendung von `g` erst nach `updateMonad s`
- `_ <-` kann entfallen, wenn Wert nicht gebraucht (`updateMonad`)

IO-Monade

- Typkonstruktor `IO`
- `return :: a -> IO a`
- `(>>=) :: IO a -> (a -> IO b) -> IO b`
- Interpretierung des Typs `IO a`: eine *Berechnung* mit Ein-/Ausgabe und Rückgabewert vom Typ `a`
- Rückgabewert einer Berechnung soll nur in einer weiteren Berechnung benutzt werden, nicht allgemein als Funktionsergebnis (es gibt eine "unsichere" vordefinierte Funktion, die das in Ausnahmefällen trotzdem ermöglicht)

I/O und Seiteneffekte

- Was bedeutet Ein/Ausgabe mit Seiteneffekten?
  1. Verlust der referenziellen Transparenz?
     `getChar` liefert nicht immer denselben Wert:

     ```
     Prelude> do { a<-getChar; b<-getChar; print (a==b) }
     12False
     ```

  2. Problem bei einer Sprache mit Lazy-Auswertung:
     Reihenfolge der Auswertung bestimmt Reihenfolge der IO-Operationen
- Lösung in Haskell:
  1. Wert von `getChar` ist nicht `Char` sondern `IO Char`, d.h.
     eine Ein/Ausgabeoperation
  2. Reihenfolge erzwungen durch Sequenz in der Monade `(>>=)`.

Vordefinierte IO-Operationen

- Lesen eines Zeichens von stdin: `getChar :: IO Char`
- Lesen einer Datei: `readFile :: FilePath -> IO String`
- Ausgabe auf stdout: `putStr :: String -> IO ()`

Viele weitere Operationen (siehe Haskell- und speziell GHC-Dokumentation)

- Dateioperationen, Dateisystem
- Zeitmessung
- Pseudozufallszahlen
- Prozesse, Signale, Nebenläufigkeit
- Aufruf externer C-Funktionen

Haskell-Hauptprogramm

Ein Haskell-Programm muss eine `main`-Funktion besitzen, um übersetzt werden zu können.

Folgendes Programm kopiert die Zeichensequenz aus der Datei ”input” in umgekehrter Reihenfolge in die Datei ”output”

```
main :: IO ()
main = do
        xs <- readFile "input"
        let ys = reverse xs
        writeFile "output" ys
```

Erweitertes Beispiel

```
f :: Integer -> Integer -> Integer
f x y = x*y

main :: IO ()
main = do
        x <- readFile "input1"
        y <- readFile "input2"
        let v = show u
            u = f (read x) (read y)
        writeFile "output" v
```

- Innerhalb eines `let`-Blocks ist Reihenfolge egal
- `read`-Funktion zum Parsen einfacher Typen
- Die Dateien `"input1"`, `"input2"` und `"output"` sollten verschieden sein

Synchronisationsprobleme (1)

Monadensequenzierung `(>>=)` erzwingt nur Reihenfolge auf oberster Ebene, keine Striktheit der beteiligten Funktionen

Dateioperationen: sind oft nicht hyperstrikt, Bsp.:

```
do
 x <- readFile "input"
 let y = map toUpper x
 writeFile "output" y
```

hier kann mit dem Schreiben von `"output"` begonnen werden bevor das Lesen von `"input"` beendet ist

Synchronisationsprobleme (2)

```
do
 tStart <- getCPUTime
 let y = f $! x
 tEnd <- getCPUTime
```

Striktheitsannotation `$!` verwenden, damit `f` überhaupt zwischen den Zeitmessungen auf `x` angewendet wird. Das Ausmaß der Auswertung wird von `f` bestimmt.

Lokale Gültigkeitsbereiche

```
main = do
          x <- return 5
          let y = x
          do
             x <- return 7
             putStr (show (x==y))    -- Ausgabe: False
          putStr (show (x==y))       -- Ausgabe: True
```

Referenzielle Transparenz bei Monaden

referenzielle Transparenz bleibt erhalten

- formale Sicht: Wert eines monadischen Ausdrucks ist nicht der Rückgabewert sondern die monadische Berechnung als solche
- Sicht des Programmierers:
  - große Ähnlichkeit zur Sequenz in imperativen Sprachen:
    kompositionelle Sicht der Monade kann verlorengehen und im Unterbewusstsein durch operationelles Denken im kleinen ersetzt werden
    Warnzeichen: lange Sequenzen im Programmtext, Tailrekursion
  - gleiche Risiken wie bei imperativer Programmierung:
    Aliasing, Redundanz, Inkonsistenz
  - Vermeidung der Risiken durch:
    monadische Kombinatoren, Zugriffsfunktionen auf den Zustand

Verlassen der Monade

- "Betreten" der Monade: `return :: a -> m a`
- Operation in der Monade: `(>>=) :: m a -> (a -> m b) -> m b`
- "Verlassen" der Monade: `? :: m a -> a`
  Problem: einbetten der Monade in das umgebende Programm
  - falls Monade "`MyM`" selbst definiert, kann auch eine Funktion `unMyM :: MyM a -> a` definiert werden
  - bei vordefinierter `IO`-Monade kein empfohlener Weg
    - wäre unsichere Operation
      - würde referenzielle Transparenz verletzen
      - Reihenfolge der Ein-/Ausgabeoperationen durch laziness kompliziert
    - nicht nötig
      - Hauptprogramm hat den passenden Typ: `main :: IO ()`
      - Anwendung monadenloser Funktionen mit Hilfe von `let`

Monaden-Kombinatoren

```
copyFile :: (String,String) -> IO ()
copyFile (x,y) = do
                   content <- readFile x
                   writeFile y content

main :: IO ()
main = do
        copyFile ("input1","output1")
        copyFile ("input2","output2")
```

Aggregation mittels mapM:

```
main :: IO ()
main = do
        mapM copyFile [ ("input"++show i,"output"++show i)
                      | i<-[1..2] ]
        return ()
```

mapM vs. foldM (1)

```
import Monad
foo :: Int -> IO Int
foo i = do putStr ("foo " ++ show i ++ " called\n")
           return (i*i)
```

```
bar :: Int -> Int -> IO Int
bar i j = do putStr ("bar " ++ show i ++ " " ++ show j ++ " called\n")
             return (i+j)
```

```
main :: IO ()
main = do
        xs <- mapM foo [1..4]
        putStr ("point A: "++show xs++"\n")
        y <- foldM bar 0 [1..4]
        putStr ("point B: "++show y++"\n")
        return ()
```

`mapM` vs. `foldM` (2)

- Sowohl `mapM` als auch `foldM` führen die durch die Liste gesteuerten Operationen sequentiell aus.
- `mapM`
  - Argument der i-ten Operation ist i-tes Element der Liste `[1..4]`
  - Ergebnis der i-ten Operation wird i-tes Element der Liste `xs`
- `foldM`
  - Argumente der i-ten Operation sind der Rückgabewert der $(i-1)$-ten Operation (bzw. Initialelement 0) und das i-te Element der Liste `[1..4]`
  - `y` ist der Rückgabewert der letzten Operation

mapM vs. foldM (3)

Ausgabe des foo/bar-Programms:

```
Main> main
foo 1 called                    -- xs <- mapM foo [1..4]
foo 2 called
foo 3 called
foo 4 called
point A: [1,4,9,16]             -- putStr ("point A: "++show xs++"\n")
-------------------------------------------------------------------------
bar 0 1 called                  -- y <- foldM bar 0 [1..4]
bar 1 2 called
bar 3 3 called
bar 6 4 called
point B: 10                     -- putStr ("point B: "++show y++"\n")
```

Typklassendeklaration Monad

Um die do-Notation für eine eigene Monade verwenden zu können, muss man sie als Instanz von Monad deklarieren

```
class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

        -- Minimal complete definition: (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

Monadengesetze

Abstraktion des ersten Arguments von `(>>=)`:

```
(>@>) :: Monad m => (a -> m b)
                 -> (b -> m c)
                 -> (a -> m c)
(f >@> g) x = f x >>= g
```

`>@>` ist assoziativ mit neutralem Element `return`:

1. `return >@> f == f`
2. `f >@> return == f`
3. `f >@> (g >@> h) == (f >@> g) >@> h`

Modularität und Erweiterbarkeit

Beispiel: Kombination von Interpretern und Hintergrundinformation

Auswahl aus zwei Interpretern

1. strikt
2. nicht-strikt

Auswahl aus drei Arten von Hintergrundinformation

1. keine: Identitätsmonade
2. Fehler: Fehlermonade
3. Operationszähler: Zustandsmonade

Strikter Interpreter (1)

```
module InterStrict where

import MonStateTrans    -- oder MonIdent oder MonError

data AExp = Var Variable
          | Const Value
          | Plus AExp AExp
          | Let Variable AExp AExp
          deriving Show

type Value    = Int
type Variable = String
type Env      = [(Variable,Value)]
```

Strikter Interpreter (2)

```
eval :: AExp -> Env -> M Value
eval (Var v) e = lookup1 v e
eval (Const c) e = return c
eval (Plus a1 a2) e = do
                      v1 <- eval a1 e
                      v2 <- eval a2 e
                      add v1 v2
eval (Let v a1 a2) e = do
                       x <- eval a1 e
                       eval a2 (update e v x)
```

Strikter Interpreter (3)

```
add :: Value -> Value -> M Value
add x y = do
            tick
            return (x+y)

update :: Env -> Variable -> Value -> Env
update e v x = (v,x):e

lookup1 :: Variable -> Env -> M Value
lookup1 v' ((v,x):e) | v'==v  = return x
                     | otherwise = lookup1 v' e
lookup1 v' _ = fail v'

test :: AExp -> String
test a = display (eval a [])
```

Modifikation Interpreter (strikt $\longrightarrow$ nicht-strikt)

- Environment umfasst Berechnungen statt Werte:

```
type Env = [(Variable, M Value)]
update :: Env -> Variable -> M Value -> Env
```

- Let-Ausdrücke werten Rumpf (noch) nicht aus:

```
eval (Let v a1 a2) e = let x = eval a1 e  -- statt x <- eval a1 e
                       in eval a2 (update e v x)
```

- Auswertung von Berechnungen beim Zugriff auf das Environment

```
lookup1 v' ((v,x):e) | v'==v = x      -- war:  return x
```

Die Identitätsmonade

```
module MonIdent where

newtype I a = I a deriving Show
instance Monad I where
  return x    = I x
  (I x) >>= f = f x
  fail i      = error "lookup failed"

type M a = I a

tick :: I ()
tick = return ()

display :: Show a => I a -> String
display (I x) = show x
```

Die Fehlermonade (1)

```
module MonError where

data E a = Error String | Ok a
         deriving Show

instance Monad E where
   return x = Ok x
   m >>= f  = case m of
                Error s -> Error s
                Ok x -> f x
   fail s = Error s
```

Die Fehlermonade (2)

```
type M a = E a

tick :: E ()
tick = return ()

display :: Show a => E a -> String
display (Error s) = "Error: "++ s
display (Ok x)    = "Ok: " ++ show x
```

Die Zustandsmonade (1)

```
module MonStateTrans where

data ST s a = ST { unST::(s -> (a,s)) }

instance Monad (ST s) where
   return x  = ST (\s -> (x,s))
   m >>= f   = ST (\s -> let (x1,s1) = unST m s
                             (x2,s2) = unST (f x1) s1
                         in (x2,s2))
   fail x = error "lookup failed"
```

Die Zustandsmonade (2)

```
type M a = ST Int a

tick :: ST Int ()
tick = ST (\s -> ((),s+1))

display :: Show a => M a -> String
display m = case unST m 0 of
              (a,s) -> "Count: " ++ show s ++ ", Value: " ++ show a
```

Test1 (Ausdruck ohne Variablen)

```
test (Plus (Const 2) (Const 3))
```

- Identitätsmonade: `"5"`
- Fehlermonade: `"Ok: 5"`
- Zustandsmonade
  - strikt: `"Count: 1, Value: 5"`
  - nicht-strikt: `"Count: 1, Value: 5"`

Test2 (Ausdruck mit undefinierter Variablen)

```
test (Plus (Const 2) (Var "x"))
```

- Identitätsmonade: `"*** Exception: lookup failed`
- Fehlermonade: `"Error: x"`
- Zustandsmonade
  - strikt: `"Count: *** Exception: lookup failed`
  - nicht-strikt: `"Count: *** Exception: lookup failed`

Test3 (Ausdruck mit definierter Variablen)

```
test (Let "x" (Plus (Const 2) (Const 3)) (Plus (Var "x") (Var "x")))
```

- Identitätsmonade: `"10"`
- Fehlermonade: `"Ok: 10"`
- Zustandsmonade
  - strikt: `"Count: 2, Value: 10"`
  - nicht-strikt: `"Count:3, Value: 10"`

Monaden-Transformer

- Kombination von Ticks, Fehlerbehandlung etc. in einer ST-Monade möglich
- besser: unabhängige Erweiterung von Monaden

im Modul `Control.Monad.Trans`:

```
class MonadTrans t where
   lift :: Monad m => m a -> t m a
```

- Monade `m` wird zur Monade `t m` erweitert
- Zugriff auf Funktionen von `t m` wie gewohnt
- Zugriff auf Funktionen von `m` mittels `lift`

Beispiel: ST $\longrightarrow$ ST+EXC (Exception)

```
newtype EXC m a = MkEXC (m (Exc a))

data Exc a = Raise Exception
           | Return a
           deriving Show

data Exception = Div0
               deriving Show
```

```
instance Monad m => Monad (EXC m) where
   return x = MkEXC (return (Return x))
   p >>= q  = MkEXC (recover p >>= r)
                where r (Raise e) = return (Raise e)
                      r (Return x) = recover (q x)

recover :: EXC m a -> m (Exc a)
recover (MkEXC g) = g

class Monad m => ExMonad m where
   raise :: Exception -> m a

instance Monad m => ExMonad (EXC m) where
   raise e = MkEXC (return (Raise e))
```

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a

instance MonadTrans EXC where
    lift g = MkEXC (do x <- g
                       return (Return x) )

type State = Int

tick :: ST State ()
tick = ST (\i -> ((),i+1))

data Exp = C Integer
         | Exp :+: Exp
         | Exp :/: Exp
         deriving Show
```

```
eval :: Exp -> EXC (ST State) Integer
eval (C i) = return i
eval (a:+:b) = do [i,j] <- mapM eval [a,b]
                  lift tick
                  return $ i+j
eval (a:/:b) = do [i,j] <- mapM eval [a,b]
                  lift tick
                  if j==0 then raise Div0
                          else return (i`div`j)

run :: Exp -> String
run exp = let (MkEXC x)    = eval exp
              (res,ticks) = unST x 0
          in ("Erg.: "++ show res ++
              ", Ticks: "++show ticks)
```

`lift`-Indirektion in der Instanzdeklaration

```
class Monad m => TickMonad m where
   tick :: m ()

instance Num m => TickMonad (ST m) where
   tick = ST (\i -> ((),i+1))

instance TickMonad m => TickMonad (EXC m) where
   tick = lift tick
```

damit:

```
eval (a:/:b) = do [i,j] <- mapM eval [a,b]
                  tick
                  if j==0 then raise Div0
                          else return (i`div`j)
```

```
> run ((C 2 :/: C 0) :+: (C 5 :/: C 3))
"Erg.: Raise Div0, Ticks: 1"

> run ((C 2 :/: C 1) :+: (C 5 :/: C 0))
"Erg.: Raise Div0, Ticks: 2"

> run ((C 2 :/: C 1) :+: (C 5 :/: C 2))
"Erg.: Return 4, Ticks: 3"

> run (C 1 :+: C 1 :+: C 1 :+: C 1 :+: C 1)
"Erg.: Return 5, Ticks: 4"
```

Kombination von Zustand und IO

Mögliche Ansätze und ihre Nachteile:

1. expliziter Zustand im IO-monadischen Programm
2. ST-Monade mit unsicheren IO-Operationen
3. Ein-/Ausgabeströme im Zustand: geringe IO-Flexibilität
4. Monaden-Schachtelungen
   (a) ST-Monade in IO-Monade: kein IO in der ST-Monade
   (b) IO-Monade in ST-Monade nicht möglich, weil IO ohne Ausgang
5. Monaden-Transformationen
   (a) ST nach IO/ST: im Prinzip dasselbe wie (1.)
   (b) IO nach IO/ST: Problem: Wiederherstellung der Zustandsinformation nach IO-Operation

Transformation IO-Monade nach IO/ST

Problem: Wiederherstellung der Zustandsinformation nach IO-Operation

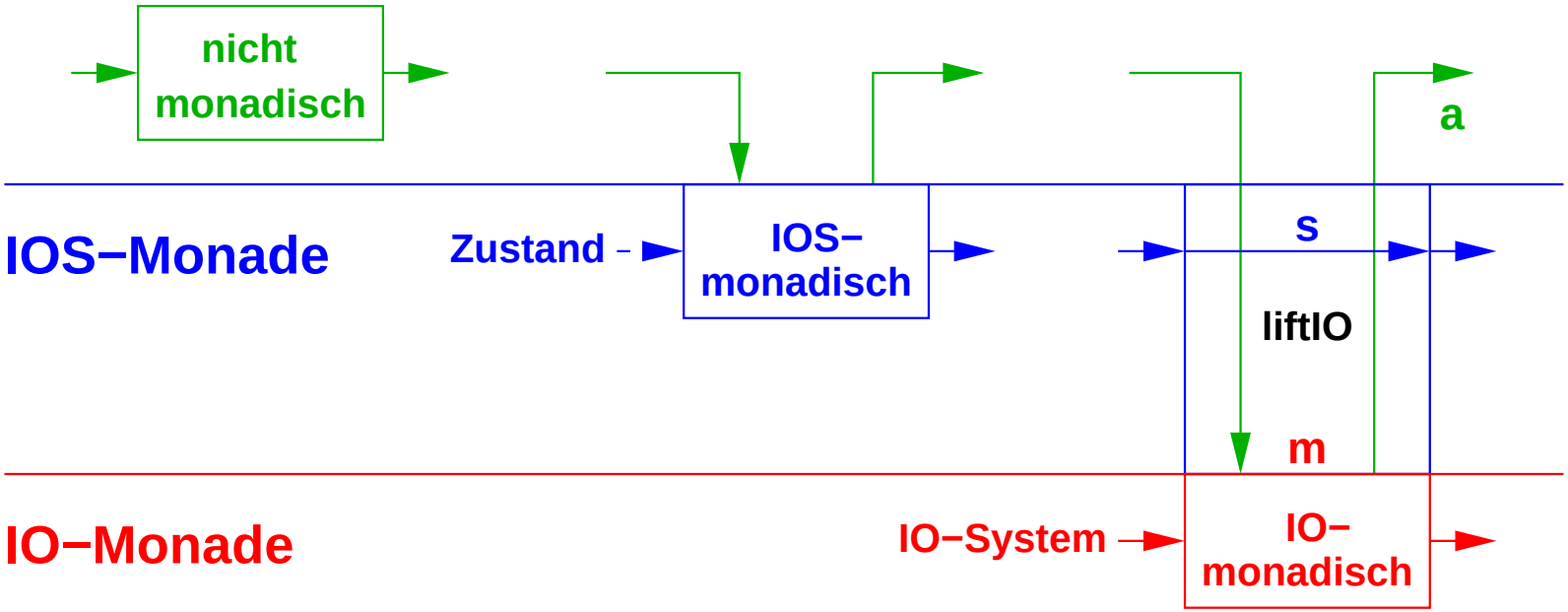
gesucht: Lösung die gleichzeitig

- Zustand im Rückgabewert der IO-Monade sichert
- IO-Operationen durchführt

Idee: betrachte Aufruf einer IO-Operation:

```
\s -> do              -- bringe Zustand s in den Scope der IO-Monade
      a <- m          -- Aufruf der IO-Operation m
      return (a,s)    -- Zustand s als Rückgabewert
```

IOS: ST-Monade mit IO




```
liftIO m = IOS (\s -> do
                        a <- m
                        return (a,s))
```

IOS: ST-Monade mit IO

```
module IOS where
import Control.Monad.Trans
-- class Monad m => MonadIO m where
--   liftIO :: IO a -> m a

data IOS s a = IOS { unIOS :: s -> IO (a, s) }

instance MonadIO (IOS m) where
   liftIO m = IOS (\s -> do a <- m
                            return (a,s))
instance Monad (IOS s) where
   return x = IOS (\s -> return (x,s))
   m >>= f  = IOS (\s -> do
                           (x,s1) <- unIOS m s
                           unIOS (f x) s1)
```

Zugriff auf den Zustand in IOS

```
data IOS s a = IOS { unIOS :: s -> IO (a, s) }

getIOS :: IOS s s
getIOS = IOS (\s -> return (s,s))

setIOS :: s -> IOS s ()
setIOS s' = IOS (\s -> return ((),s'))

modIOS :: (s->s) -> IOS s s
modIOS f = IOS (\s -> return (s, f s))
```

Abstrakte Vorstellung von der IOS-Monade

Die IOS-Monade ist ein abstrakter Datentyp mit zwei Zugriffsarten

1. Zustandsoperation `f` definiert im ADT
   (a) nur lesend: `f :: IOS s t`
   (b) auch schreibend: `f :: s -> IOS s t`
2. Ein-Ausgabeoperation `g :: IO a`
   - bei Funktionen `h :: b0 -> ... -> bn -> IO a`
     wende erst alle Argumente an, dann hat Ergebnis den Typ `IO a`
   - Einbettung durch `liftIO`, wobei `liftIO g :: IOS a`

Kleines Anwendungsbeispiel (1)

```
module Main where

import IO   -- system definitions
import IOS  -- own definition

type State = [String]

type M = IOS State

queryUser :: String -> M String
queryUser s = liftIO $ do
                       putStr $ "please enter text " ++ s ++ ": "
                       getLine
```


Kleines Anwendungsbeispiel (2)

```
mainIOS :: M ()
mainIOS = do
            x <- queryUser "1"
            setIOS [x]
            y <- queryUser "2"
            modIOS (++[y])
            z <- getIOS
            liftIO $ putStrLn ("input was " ++ show z)
            return ()

main :: IO ()
main = do
         mapM (flip hSetBuffering NoBuffering) [stdin,stdout]
         unIOS mainIOS []
         return ()
```

Kleines Anwendungsbeispiel (3)

```
> ghc --make ExampleIOS.hs
Chasing modules from: ExampleIOS.hs
Compiling IOS                ( IOS.hs, ./IOS.o )
Compiling Main               ( ExampleIOS.hs, ./ExampleIOS.o )
Linking ...

> a.out
please enter text 1: Hello
please enter text 2: world
input was ["Hello","world"]
```

Simpler Texteditor (1)

```
module Main where
import IO
import Monad
import IOS

data State = St { line::Int, text::[String] }

type M = IOS State

untilM :: Monad m => (a->Bool) -> m a -> m ()
untilM pred body = loop
 where loop = do
                y <- body
                unless (pred y) loop
```

Simpler Texteditor (2)

```
editText :: M ()
editText =
   untilM (== ":bye")
          (do
            state <- getIOS
            s <- liftIO $ do putStr (show (line state) ++ " ")
                             getLine
            let newLine = if (take 5 s == ":back")
                           then read (drop 5 s)
                           else line state + 1
            modIOS (\state -> state {line = newLine,
                                     text = take newLine (text state
                                                          ++ [s])})
            return s)
```

Simpler Texteditor (3)

```
mainIOS :: M ()
mainIOS = do
            editText
            state <- getIOS
            liftIO $ putStrLn ("input was " ++
                               show (init (text state)))
            return ()

main :: IO ()
main = do
         mapM (flip hSetBuffering NoBuffering) [stdin,stdout]
         unIOS mainIOS (St {line=0, text=[]})
         return ()
```

Simpler Texteditor (4)

```
> a.out
0 the
1 quick
2 bwron
3 fox
4 :back2
2 brown
3 fox
4 :bye
input was ["the","quick","brown","fox"]
```