

Einführung in die Funktionale Programmierung (EFP):

IO in Haskell und Monadisches Programmieren

Prof. Dr. Manfred Schmidt-Schauß

WS 2023/24

Stand der Folien: 6. Februar 2024

Ziele des Kapitels

- Was sind Monaden / Monadische Programmierung?
- Programmierung: Ein- und Ausgabe in Haskell
- Zustandsbasiertes Programmieren mit Monaden allgemein
- (Monad-Transformer: „Vereinigung“ mehrerer Monaden)

Übersicht

- 1 Monadisches Programmieren
- 2 Weitere Monaden
- 3 I/O in Haskell
 - Einleitung
 - Monadisches IO
 - IO Verzögern
 - Speicherplätze
 - Kontrollstrukturen

Monadisches Programmieren

- **Monadisches Programmieren** (Idee von **Phil Wadler**)
= **Strukturierungsmethode**, um **sequentiell** ablaufende Programme **lazy** bzw. **call-by-need** und funktional zu implementieren
- **Ein-Ausgabe in Haskell** wird in `Main` angesteuert mittels monadischem Ausdruck.
- Begriff **Monade** entstammt der Kategorientheorie (Teilgebiet der Mathematik: Morphismen, Isomorphismen, ...)
- Für die Programmierung:
Monade ist ein **Typ**(-konstruktor) + **Operationen**, wobei die sog. **monadischen Gesetze** gelten
- In **Haskell**: Umsetzung von Monaden durch die **Typklasse Monad**.
Jeder Datentyp, der Instanz der Klasse **Monad** ist und die Gesetze erfüllt, ist eine **Monade**

Monadisches Programmieren: Beispiel

So sieht ein einfaches Beispielprogramm am Ende aus:

```
main :: IO ()
main = do
  y <- getLine
  putLine y
  putChar ('\n')
  putLine ("Weitermachen? Y/N")
  putChar ('\n')
  x <- getChar
  if x == 'Y' || x == 'y'
  then do
    putChar ('\n')
    main
  else putLine ("Ende")
```

Sieht sehr imperativ aus; **Ist auch imperativ in dieser Schicht**

Monadisches Programmieren

Beispiele von Monaden in Haskell:

- Sichtbar in Haskell: do-Notation (kommt noch)
- List Comprehensions:
 - [(x,y) | x <- [1..10], y <- [x..10], (x,y) /= (10,10)]
 - Reihenfolge der Abarbeitung wird festgelegt
 - Weitergabe von Werten über Bindungen
 - Rückgabewert
- Monaden sind notwendig beim Zusammenwirken von IO-Befehlen und **imperativem (sequentiellen) Programmieren**, und Haskell's purer funktionaler Programmierung.

Die Typklasse Monad

```
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

m >> k   = m >>= \_ -> k
fail s   = error s
```

- Idee: man setzt Kommandos aus anderen zusammen.
- Monad ist eine **Konstruktorklasse**, da m ein Typkonstruktor ist.
- Instanzen müssen nur >>= und return implementieren.
- Ein Objekt vom Typ m a heißt auch **monadische Aktion**
- Strukturierungsprinzip:
 - Setze neue Aktionen
 - aus kleineren Aktionen zusammen (sequentiell).
 - Mittels >>= oder >>

Die Typklasse Monad (2)

Die Operatoren:

- **return** :: a -> m a
verpackt beliebigen Wert (Ausgabe) als monadische Aktion
- **(>>)** :: m a -> m b -> m b
 - heißt „then“
 - verknüpft zwei monadische Aktionen **sequentiell** zu einer Neuen
 - zweite Aktion verwendet **nicht** das Ergebnis der ersten
- **(>>=)** :: m a -> (a -> m b) -> m b
 - heißt „bind“
 - verknüpft zwei monadische Aktionen **sequentiell** zu einer Neuen
 - zweite Aktion kann Ergebnis der ersten benutzen.

Die Typklasse Monad (2b)

>> kann mit >>= implementiert werden:

```
m >> k = m >>= \_ -> k
```

Aber: >> kann **nicht parallel** implementiert werden.
Denn: der Zustand wird verändert.

Parallele Aktionen: ?????

Man müsste dazu die Monade selbst erweitern,
zu einer Monade mit parallelen Aktionen.

Problem: Es ist nicht (automatisch) festgelegt (und nicht
verifizierbar)
ob die Aktionen wirklich parallel sind.

Eine Antwort darauf: **Concurrent Haskell** (hier nicht behandelt)

Beispiel: Datentyp Maybe

```
data Maybe a = Nothing | Just a
```

- ermöglicht Unterscheidung zwischen „fehlerhaften“ und „erfolgreichen“ Berechnungen

- Maybe-Verwendung als Monade:

Sequenz von Berechnungen

$$S = e_1 \gg= \dots \gg= e_n$$

Wenn in der Sequenz die Berechnung von e_i fehlschlägt
(Nothing liefert), dann schlägt die gesamte Berechnung von
 S fehl (liefert Nothing)

Beispiel: Datentyp Maybe (2)

Maybe-Instanz für Monad:

```
instance Monad Maybe where
  return      = Just
  fail s      = Nothing
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

Die Typen sind:

```
return :: a -> Maybe a
fail   :: String -> Maybe a
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Beispiel: Verwendung der Maybe-Monade

Zwei „Datenbanken“ (Listen):

- (Kontonummer, Kontostand):
db = [(101,100), (102,200), (103,-500)]
- (Kontonummer, Passwörter) sind gespeichert, z.B.
passdb = [(101,"KSPW!3"), (102,"0w23="), (103,"12ko12")]
- Implementiere:** getKontostand: liefert Kontostand bei
richtigem Passwort

```
getKontostand knr pass =
  lookup knr db >>=
    \x -> lookup knr passdb >>=
      \y -> eqMaybe pass y >>
        return x

eqMaybe a b
  | a == b = Just True
  | otherwise = Nothing
```

Beispiel: Verwendung der Maybe-Monade (2)

Beispielaufufe

```
getKontostand 101 "KSPW!3" ----> Just 100
getKontostand 102 "KSPW!3" ----> Nothing
getKontostand 10  "KSPW!3" ----> Nothing
```

Gleiche Funktionalität ohne Monaden:

```
getKontostand knr pass =
  case lookup knr db of
    Nothing -> Nothing
    Just x -> case lookup knr passdb of
      Nothing -> Nothing
      Just y -> case eqMaybe pass y of
        Nothing -> Nothing
        Just _ -> Just x
```

Funktioniert, aber unübersichtlich!

do-Notation: Beispiel

Vorher:

```
getKontostand knr pass =
  lookup knr db >>=
    \x -> lookup knr passdb >>=
      \y -> eqMaybe pass y >>
        return x
```

Mit **do-Notation**:

```
getKontostand knr pass =
  do
    x <- lookup knr db
    y <- lookup knr passdb
    eqMaybe pass y
    return x
```

do-Notation (1)

- Einleitung eines do-Blocks durch **do**

Folge von "Anweisungen" im Rumpf, 1 Anweisung ist:

- **Monadische Aktion**, oder
- **x <- aktion**
damit kann auf das Ergebnis der Aktion zugegriffen werden
(x gültig ab der nächsten Anweisung)
- **let x = a**
zur Definition **funktionaler** Variablen (Achtung: kein in!)

Klammern + Semikolons oder Layout mit Einrückung:

```
do
  a1
  a2
  ...
  an
```

do {a1; a2;...; an} gleich zu

do-Notation (2)

do-Notation ist nur **syntaktischer Zucker**.

Übersetzung nach Haskell ohne **do**:

```
do { x <- e ; s } = e >>= (\x -> do { s })
do { e ; s }      = e >> do { s }
do { e }          = e
do { let binds; s } = let binds in do { s }
```

Beispiel: Übersetzung von do

```
getKontostand knr pass =
  do
    x <- lookup knr db
    y <- lookup knr passdb
    eqMaybe pass y
    return x
```

```
getKontostand knr pass =
  lookup knr db >>=
    \x ->
      do
        y <- lookup knr passdb
        eqMaybe pass y
        return x
```

```
getKontostand knr pass =
```

M. Schmidt-Schauß

IO in Haskell und Monadisches Programmieren

17 / 74

```
\x ->
```

```
do
```

Monadisches Programmieren Weitere Monaden I/O in Haskell

Monadische Gesetze (2)

- Gelten die 3 Gesetze, so ist die Eigenschaft "ist Monade" erfüllt
- Dann gilt: Die Auswertung der Aktionen wird **sequentiell** durchgeführt.

Monadische Gesetze

Erstes Gesetz: „return ist links-neutral bzgl. >>=“

```
return x >>= f = f x
```

Zweites Gesetz: „return ist rechts-neutral bzgl. >>=“

```
m >>= return = m
```

Drittes Gesetz: „eingeschränkte Assoziativität von >>=“

```
m1 >>= (\x -> m2 x >>= m3)
```

=

```
(m1 >>= m2) >>= m3 wobei x ∉ FV(m2, m3)
```

Ungenauigkeit: Was genau bedeutet das = -Zeichen?

M. Schmidt-Schauß

IO in Haskell und Monadisches Programmieren

18 / 74

Monadisches Programmieren Weitere Monaden I/O in Haskell

Maybe erfüllt die monadischen Gesetze

```
instance Monad Maybe where
  return      = Just
  fail s      = Nothing
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

1. Gesetz: lässt sich direkt nachrechnen:

```
return x >>= f
= Just x >>= f
= f x
```

Wir setzen voraus, dass bereits gezeigt ist, dass Reduktionen die Gleichheit erhalten!

Maybe erfüllt die monadischen Gesetze

```
instance Monad Maybe where
  return      = Just
  fail s      = Nothing
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

2. Gesetz: Fallunterscheidung:

Fall 1: m wertet zu Nothing aus:

```
Nothing >>= return = Nothing
```

Fall 2: m wertet zu Just s aus:

```
Just s >>= return = return s = Just s
```

Fall 3: Die Auswertung von m terminiert nicht.

Dann terminiert auch die Auswertung von `m >>= return` nicht.

Maybe erfüllt die monadischen Gesetze (3)

```
instance Monad Maybe where
  return      = Just
  fail s      = Nothing
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

3. Gesetz: $(m1 \gg= m2) \gg= m3 = m1 \gg= (\lambda x \rightarrow m2\ x \gg= m3)$

Fall 1: m1 wertet zu Nothing aus

```
(Nothing >>= m2) >>= m3 = Nothing >>= m3
= Nothing = Nothing >>= (\x -> m2 x >>= m3)
```

Fall 2: m1 wertet zu Just e aus

```
(Just e >>= m2) >>= m3 = m2 e >>= m3
= (\x -> m2 x >>= m3) e = Just e >>= (\x -> m2 x >>= m3)
```

Fall 3: Die Auswertung von m1 divergiert:

dann divergieren sowohl $(m1 \gg= m2) \gg= m3$ als auch $m1 \gg= (\lambda x \rightarrow m2\ x \gg= m3)$.

Die Listen-Monade

Auch Listen sind Instanz der Klasse Monad:

```
instance Monad [] where
  m >>= f = concatMap f m
  return x = [x]
  fail s = []
  --[] bedeutet Listentyp
  --[] bedeutet nil
```

Die Implementierung von `>>=` und `return` entspricht dabei den List Comprehensions.

Z.B. `[x*5 | x <- [1,2,3]]`

```
[1,2,3] >>= \x -> [x*5]
= concatMap (\x -> [x*5]) [1,2,3]
= concat [[1*5], [2*5], [3*5]]
= [1*5, 2*5, 3*5]
= [5, 10, 15]
```

Die Listen-Monade (2)

Deutlicher mit do-Notation:

Beispiel: `[x*y | x <- [1..10], y <- [1,2]]`

kann man mit der Listenmonade (do-Notation hier) schreiben wie folgt; Es ist auch in dieser Form ausführbar (gleiche Ausführung)

```
do
  x <- [1..10]
  y <- [1,2]
  return (x*y)
```

In beiden Fällen erhält man

```
[1,2,2,4,3,6,4,8,5,10,6,12,7,14,8,16,9,18,10,20]
```

D.h.: Baumartige sequentielle Auswertung passt auch zu monadischen Aktionen

Die Listen-Monade (2)

Beispiel für Filter in der `do`-Notation:

```
[x*y | x <- [1..10], y <- [1,2], x*y < 15]
```

kann man mit der Listenmonade (do-Notation hier) schreiben als:

```
do
  x <- [1..10]
  y <- [1,2]
  if (x*y < 15) then return undefined else fail ""
  return (x*y)
```

Man erhält in beiden Fällen:

```
[1,2,2,4,3,6,4,8,5,10,6,12,7,14,8,9,10]
```

Es gilt: Die Monad-Instanz für Listen **erfüllt die monadischen Gesetze**

(ohne Beweis, da uns hierfür Methoden fehlen:
Induktionserweiterung für unendliche Listen)

Die StateTransformer-Monade

- StateTransformer = Datentyp, der die **Veränderung eines Zustands** kapselt.
- D.h. eine Funktion von Zustand zu Zustand
- Dabei kann noch ein Ausgabe-Wert berechnet werden.
- Ist polymorph über dem Zustand und dem Wert

```
data StateTransformer state a = ST (state -> (a,state))
```

Die StateTransformer-Monade (2)

```
data StateTransformer state a = ST (state -> (a,state))
```

Monad-Instanz:

```
instance Monad (StateTransformer s) where
  return a      = ST (\s -> (a,s))
  (ST x) >>= f  = ST (\s -> case x s of
                              (a,s') -> case (f a) of
                                          (ST y) -> (y s'))
```

- Mit `>>=` kann man aus kleineren StateTransformer-Aktionen neue größere zusammensetzen

Haskell ab Version 10: braucht auch Instanzen von Functor und Applicative: siehe:

<https://wiki.haskell.org/>

Functor-Applicative-Monad-Proposal#Applying-the_AMP_to_GHC_and_then_Haskell_in_practice

Beispiel: Taschenrechner

- Einfacher Taschenrechner, der ...
- Bei jeder Operation direkt ausgewertet
- Nur die Grundrechenarten beherrscht

Innerer Zustand des Taschenrechners:

(Funktion, Zahl)

- Die Funktion enthält aktuelles Zwischenergebnis plus Operation
- Zahl: Die aktuelle erkannte (eingegabene Zahl)

Beispiel: Taschenrechner (2)

Beispiel: Abarbeitung von 30+50= und innere Zustände:

Zustand	Resteingabe
(\x -> x, 0.0)	30+50=

Zustand	Resteingabe
(\x -> x, 0.0)	30+50=
(\x -> x, 3.0)	0+50=

Zustand	Resteingabe
(\x -> x, 3.0)	0+50=
(\x -> x, 30.0)	+50=

M. Schmidt-Schauß

IO in Haskell und Monadisches Programmieren

29 / 74

Beispiel: Taschenrechner (4)

- CalcTransformer die nur den inneren Zustand ändern, geben keinen Wert zurück
- Modellierung: Nulltupel () als Rückgabewert

Zustandsveränderung für bel. binären Operator:

```
oper :: (Float -> Float -> Float) -> CalcTransformer ()
oper op = ST $ \ (fn,zahl) -> ((), (op (fn zahl), 0.0))
```

clear: Löschen der letzten Eingabe:

```
clear :: CalcTransformer ()
clear = ST $ \ (fn,zahl) ->
  ((), if zahl == 0.0
       then startState
       else (fn, 0.0))
```

M. Schmidt-Schauß

IO in Haskell und Monadisches Programmieren

31 / 74

(\x -> x, 3.0)	0+50=
----------------	-------

Beispiel: Taschenrechner (3)

Typsynonym für den Zustand:

```
type CalcState = (Float -> Float, Float)
```

Startzustand:

```
startState = (id, 0.0)
```

StateTransformer mit CalcState als Zustand:

```
type CalcTransformer a = StateTransformer CalcState a
```

Nächste Aufgabe:

Implementiere CalcTransformer passend zu den Funktionen des Taschenrechners

M. Schmidt-Schauß

IO in Haskell und Monadisches Programmieren

30 / 74

Beispiel: Taschenrechner (5)

total: Ergebnis berechnen:

```
total :: CalcTransformer ()
total = ST $ \ (fn,zahl) -> ((), (id, fn zahl))
```

digit: Eine Ziffer verarbeiten:

```
digit :: Int -> CalcTransformer ()
digit i = ST $ \ (fn,zahl) -> ((), (fn, (fromIntegral i) + zahl*10.0))
```

readResult: Ergebnis auslesen:

```
readResult :: CalcTransformer Float
readResult = ST $ \ (fn,zahl) -> (fn zahl, (fn,zahl))
```

M. Schmidt-Schauß

IO in Haskell und Monadisches Programmieren

32 / 74

Beispiel: Taschenrechner (6)

calcStep: Ein Zeichen der Eingabe verarbeiten:

```
calcStep :: Char -> CalcTransformer ()
calcStep x
  | isDigit x = digit (fromIntegral $ digitToInt x)

calcStep '+' = oper (+)
calcStep '-' = oper (-)
calcStep '*' = oper (*)
calcStep '/' = oper (/)
calcStep '=' = total
calcStep 'c' = clear
calcStep _   = ST $ \ (fn,z) -> ((), (fn,z))
```

Beispiel: Taschenrechner (7)

calc: Hauptfunktion, monadische Abarbeitung:

```
calc []      = readResult
calc (x:xs) = do
    calcStep x
    calc xs
```

Ausführen des Taschenrechners:

```
runCalc :: CalcTransformer Float -> (Float, CalcState)
runCalc (ST akt) = akt startState

mainCalc xs = fst (runCalc (calc xs))
```

Beispiel: Taschenrechner (8)

Beispiele:

```
mainCalc "1+2*3"      ---> 9.0
mainCalc "1+2*3="     ---> 9.0
mainCalc "1+2*3c*3"   ---> 0.0
mainCalc "1+2*3c5"    ---> 15.0
mainCalc "1+2*3c5====" ---> 15.0
```

Es fehlt noch:

Sofortiges Ausdrucken nach Eingabe
⇒ Interaktion und I/O (gleich))

Ein- und Ausgabe in Haskell

Problematik

Echte Seiteneffekte sind in purem Haskell nicht erlaubt.

Warum?

Annahme: `getZahl :: Int` wäre eine „Funktion“, die eine Zahl von der Standardeingabe liest

Referentielle Transparenz

Gleiche Funktion angewendet
auf gleiche Argumente liefert stets
das gleiche Resultat

Referentielle Transparenz ist verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.

! Das ist unverträglich mit Haskell

(Semantik, Compilierung, Auswertung, Optimierung)

Problematik (2)

Gelten (bei Seiteneffekten) noch mathematische (arithmetische) Gleichheiten wie

$$x + x = 2 * x ?$$

Nein: für $x = \text{getZahl}$ z.B.

`getZahl+getZahl ---> 1+getZahl ---> 1+2 ---> 3`

aber:

`2*getZahl ---> gerade Zahl`

Problematik (3)

Weiteres Problem: Betrachte die Auswertung von

`length [getZahl, getZahl]`

Wie oft wird nach einer Zahl gefragt?

```
length (getZahl:(getZahl:[]))
---> 1 + length (getZahl:[])
---> 1 + (1 + (length []))
---> 1 + (1 + 0)
---> 1 + 1
---> 2
```

⇒ Keine Fragen gestellt!, da `length` die Auswertung der Listen-Elemente nicht braucht

⇒ Festlegung auf eine genaue Auswertungsreihenfolge nötig.
(Nachteil: verhindert Optimierungen + Parallelisierung)

Monadisches I/O

Kapselung des I/O

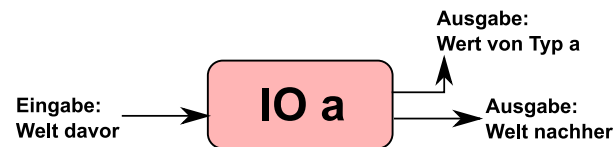
- Datentyp `IO a`
- Wert des Typs `IO a` ist eine I/O-Aktion, die beim Ausführen Ein-/Ausgaben durchführt und anschließend einen Wert vom Typ `a` liefert.
- `IO` ist Instanz von `Monad`
- Analog zu `StateTransformer`, wobei aus Haskell-Sicht: der Zustand ist außerhalb des Programms (**Welt**).
- Programmieren in Haskell I/O-Aktionen, Ausführung quasi außerhalb von Haskell

Monadisches I/O (2)



Vorstellung:

```
type IO a = Welt -> (a, Welt)
```



Monadisches I/O (3)

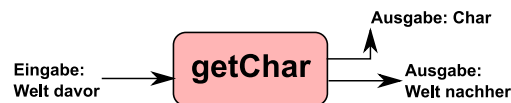


- Werte vom Typ `IO a` sind **Werte**, d.h. sie können nicht weiter **ausgewertet** werden
- Sie können allerdings **ausgeführt** werden (als Aktion)
- diese operieren auf einer Welt als Argument
- Ein Wert vom Typ `IO a` kann nicht zerlegt werden durch Pattern-Matching, da der Datenkonstruktor versteckt ist (bzw. nicht existiert). Das ist Absicht, und anders geht es auch nicht.
- Es gibt vorgegebene primitive IO-Aktionen!

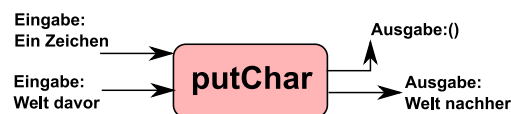
Primitive I/O-Operationen



```
getChar :: IO Char
```



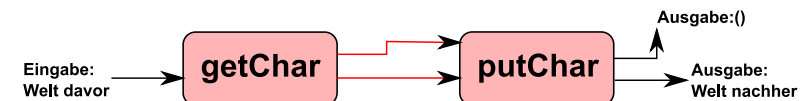
```
putChar :: Char -> IO ()
```



I/O Aktionen programmieren



- Man braucht Operationen, um I/O Operationen miteinander zu kombinieren!
- Z.B. erst ein Zeichen lesen (`getChar`), danach dieses Zeichen ausgeben (`putChar`)

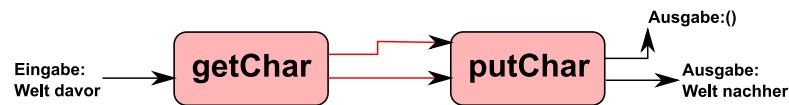


I/O Aktionen komponieren

Die gesuchte Verknüpfung bietet der `>>=` Operator
(die entsprechende Typinstanz)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

```
echo :: IO ()
echo = getChar >>= putChar
```



```
echo = do
  c <- getChar
  putChar c
```

Alternativ mit do:

I/O Aktionen komponieren (3)

Beispiel mit `>>`:

```
(>>) :: IO a -> IO b -> IO b
```

```
echoDup :: IO ()
echoDup = getChar >>= (\x -> putChar x >> putChar x)
```

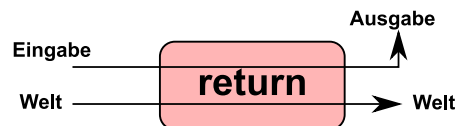
Alternativ mit do:

```
echoDup = do
  x <- getChar
  putChar x
  putChar x
```

I/O Aktionen komponieren (4)

I/O-Aktionen zusammenbauen, die zwei Zeichen liest und als Paar zurück liefert

```
getTwoChars :: IO (Char,Char)
getTwoChars = do
  x <- getChar
  y <- getChar
  return (x,y)
```



Beispiel

Eine Zeile einlesen

```
getLine :: IO [Char]
getLine = do c <- getChar;
  if c == '\n' then
    return []
  else
    do
      cs <- getLine
      return (c:cs)
```

Implementierung der IO-Monade

```
newtype IO a = IO (Welt -> (a,Welt))

instance Monad IO where
  (IO m) >>= k = IO (\s -> case m s of
    (a',s') -> case (k a') of
      (IO k') -> k' s'
  return x = IO (\ s -> (x, s))
```

Interessanter Punkt:

Implementierung ist nur richtig bei **call-by-need** Auswertung

Implementierung der IO-Monade (2)

Beispiel von Simon Peyton Jones:

```
getChar >>= \c -> (putChar c >> putChar c)
```

wird übersetzt in (Vereinfachung: ohne IO-Konstruktor)

```
\w -> case getChar w of
  (c,w1) -> case putChar c w1 of
    (_,w2) -> putChar c w2
```

Wenn beliebiges Kopieren korrekt wäre (a la call-by-name):

```
\w -> case getChar w of
  (c,w1) -> case putChar c w1 of
    (_,w2) -> putChar (fst (getChar w)) w2
```

Nun 2 Probleme:

- `getChar w` wird zwei Mal aufgerufen
- Der Weltzustand `w` wurde verdoppelt

Deshalb:

- Implementierung nur korrekt, wenn nicht beliebig kopiert (oder gemergt) wird.
- GHC führt keine solche Transformation durch.

Monadische Gesetze und IO

Oft liest man:

IO ist eine Monade.

D.h. die monadischen Gesetze sind erfüllt.

Aber:

```
(return True) >>= (\x -> undefined) ≠ (\x -> undefined) True
```

```
seq ((return True) >>= (\x -> undefined)) False ----> False
seq ((\x -> undefined) True) False ----> undefined
```

Wenn man den Gleichheitstest nur auf Werte (ohne Kontext wie `(seq [] False)`) beschränkt, dann gelten die Gesetze vermutlich.

Wenn man `seq` auf nicht-monadische Ausdrücke beschränkt, dann gelten die Gesetze.

GHC: ignoriert das Problem.

Monadisches I/O: Anmerkungen

- Beachte: Es gibt „keinen Weg aus der Monade heraus“
- Aus I/O-Aktionen können nur I/O-Aktionen zusammengesetzt werden
- Keine Funktion vom Typ `IO a -> a!`
- Wenn obiges gilt, funktioniert I/O sequentiell
- Das ist nur die halbe Wahrheit!
- Aber: Man möchte auch “lazy I/O”
- Modell passt dann eigentlich nicht mehr

Beispiel: readFile

readFile: Liest den Dateinhalt aus
– explizit mit **Handles** (= erweiterte Dateizeiger)

```
-- openFile :: FilePath -> IOMode -> IO Handle
-- hGetChar :: Handle -> IO Char

readFile :: FilePath -> IO String
readFile path =
  do
    handle <- openFile path ReadMode
    inhalt <- leseHandleAus handle
    return inhalt
```

Beispiel: readFile (2)

Handle auslesen: Erster Versuch

```
leseHandleAus handle =
  do
    ende <- hIsEOF handle
    if ende then hClose handle >> return []
    else
      do
        c <- hGetChar handle
        cs <- leseHandleAus handle
        return (c:cs)
```

Ineffizient: **Komplette** Datei wird gelesen, **bevor** etwas zurück gegeben wird.

```
*Main> readFile "LargeFile" >>= print . head
'1'
7.09 secs, 263542820 bytes
```

unsafeInterleaveIO

unsafeInterleaveIO :: IO a -> IO a

- bricht strenge Sequentialisierung auf
- gibt sofort etwas zurück **ohne** die Aktion auszuführen
- Aktion wird “by-need” ausgeführt: erst wenn die Ausgabe vom Typ a in IO a benötigt wird.
- nicht vereinbar mit “Welt”-Modell!

Handle auslesen: verzögert

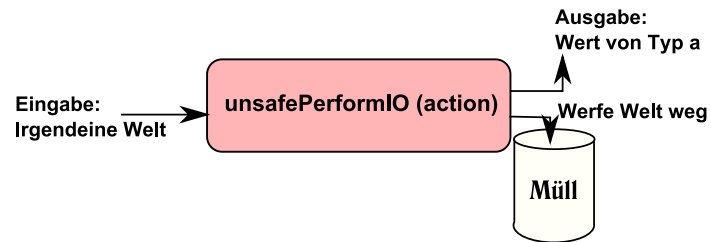
```
leseHandleAus handle =
  do
    ende <- hIsEOF handle
    if ende then hClose handle >> return []
    else
      do
        c <- hGetChar handle
        cs <- unsafeInterleaveIO (leseHandleAus handle)
        return (c:cs)
```

Test:

```
*Main> readFile1 "LargeFile" >>= print . head}
'1'
(0.00 secs, 0 bytes)
```

UnsafePerformIO

- `unsafePerformIO :: IO a -> a`
- unsauberer Sprung aus der Monade



Nicht im Haskell-Standard enthalten

Implementierung von unsafeInterleaveIO

```
unsafeInterleaveIO :: IO a -> IO a
unsafeInterleaveIO a = return (unsafePerformIO a)
```

- Führt Aktion direkt mit neuer Welt aus
- Neue Welt wird verworfen
- Das Ganze wird mit `return` wieder in die IO-Monade verpackt

Veränderliche Speicherplätze



Nur IO-monadisch verwendbar;
Mit polymorphem Typ des Inhalts:

```
data IOREf a -- Abstrakter Typ

newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Imperativ (z.B. C): **In Haskell mit IOREfs**

```
int x := 0      do
x := x+1       x <- newIORef 0
               y <- readIORef x
               writeIORef x (y+1)
```

Kontrollstrukturen



- Monadisches Programmieren ähnelt der imperativen Programmierung
- Daher auch möglich: Kontrollstrukturen wie Schleifen, Sprünge etc. zu definieren
- Wir betrachten Schleifen als Beispiele

Schleifen (1)

Repeat-until-Schleife:

```
repeatUntil :: (Monad m) => m a -> m Bool -> m ()
repeatUntil koerper bedingung =
  do koerper
    b <- bedingung
    if b then return () else repeatUntil koerper bedingung
```

Beispiel: Imperatives Programm / Monadisches Programm

```
X := 0;
repeat
  print X;
  X := X+1;
until X > 100

dieErstenHundertZahlen =
  do
    x <- newIORef 0
    repeatUntil
      (do
        wertVonX <- readIORef x
        print wertVonX
        writeIORef x (wertVonX + 1)
      )
    (readIORef x >= \x -> return (x > 100))
```

Schleifen (2)

While-Schleife

```
while :: (Monad m) => m Bool -> m a -> m ()
while bedingung koerper =
  do b <- bedingung
    if b then do
      koerper
      while bedingung koerper
    else return ()
```

Beispiel: Imperatives Programm / Monadisches Programm

```
X := 0;
while X <= 100 do
  print X;
  X := X+1;

dieErstenHundertZahlen' =
  do
    x <- newIORef 0
    while (readIORef x >= \x -> return (x <= 100))
      (do
        wertVonX <- readIORef x
        print wertVonX
        writeIORef x (wertVonX + 1)
      )
```

Nützliche Funktionen für Monaden (1)

- Einige nützliche monadische Funktionen
- Die meisten sind in `Control.Monad` vordefiniert

Aktion endlos wiederholen: `forever`:

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

`when`: wie ein imperatives `if-then` (ohne `else`-Zweig).

```
when :: (Monad m) => Bool -> m () -> m ()
when p s = if p then s else return ()
```

Nützliche Funktionen für Monaden (2)

`sequence`: Monadische Aktionen einer Liste sequentiell durchführen

```
sequence :: (Monad m) => [m a] -> m [a]
sequence [] = return []
sequence (action:as) = do
  r <- action
  rs <- sequence as
  return (r:rs)
```

`sequence` ohne Ergebnisse: `sequence_`

```
sequence_ :: (Monad m) => [m a] -> m ()
sequence_ [] = return ()
sequence_ (action:as) = do
  action
  sequence_ as
```

Nützliche Funktionen für Monaden (3)

map-Ersatz für die Monade: mapM

```
mapM      :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
```

Analog dazu mapM_: Ergebnisse verwerfen

```
mapM_     :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Z.B.

```
mapM_ print [1..100]
----> sequence_ [print 1, print 2,... print 100]
----> print 1 >> print 2 >> ... >> print 100
(druckt die Zahlen von 1 bis 100)
```

```
mapM (\_ -> getChar) [1..9]
----> sequence [getChar, ... getChar]
----> getChar >>= \c1 -> getChar >>= \c2 -> ... return [c1,c2,...,c9]
(liest 9 Zeichen und gibt sie als Liste zur"uck)
```

Zurück zum Taschenrechner

Der bisherige Taschenrechner hat kein I/O durchgeführt.

Wünschenswert:

- Eingegebene Zeichen werden sofort verarbeitet
- Ausgabe erscheint sofort

Man hätte gerne:

```
calc = do c <- getChar
         if c /= '\n' then do
             calcStep c
             calc
         else return ()
```

Funktioniert nicht!:

`getChar` ist in der IO-Monade, aber
`calcStep` ist in der StateTransformer-Monade!

Monad-Transformer

- **Lösung:** Verknüpfe zwei Monaden zu einer neuen
- Genauer: Erweitere die StateTransformer-Monade, so dass "Platz" für eine weitere Monade ist.
- Andere Sicht:
Neue Monade = IO-Monade erweitert mit Zustandsmonade

```
-- alt:
newtype StateTransformer state a =
    ST (state -> (a,state))

-- neu:
newtype StateTransformerT monad state a =
    STT (state -> monad (a,state))
```

- Die gekapselte Funktion ist jetzt eine **monadische Aktion**.
- Monaden, die um eine Monade erweitert sind, nennt man **Monad-Transformer**

StateTransformerT

Monaden-Instanz für StateTransformerT:

```
instance Monad m => Monad (StateTransformerT m s) where
    return x = STT $ \s -> return (x,s)
    (STT x) >>= f = STT $ (\s -> do
        (a,s') <- x s
        case (f a) of
            (STT y) -> (y s'))
```

Taschenrechner (1)

Typsynonym

```
type CalcTransformerT a = StateTransformerT IO CalcState a
```

⇒ `CalcTransformerT` ist ein `StateTransformerT` mit
`IO`-Monade und `CalcState` als Zustand

Taschenrechner (2)

Nächste Aufgabe: “Lifte” die Funktionen für `CalcTransformer` in
den Typ `CalcTransformerT`

Solange diese kein `IO` durchführen ist das einfach:

```
lift :: CalcTransformer a -> CalcTransformerT a
lift (ST fn) = STT $ \x -> return (fn x)
```

```
oper' :: (Float -> Float -> Float) -> CalcTransformerT ()
oper' op = lift (oper op)
```

```
digit' :: Int -> CalcTransformerT ()
digit' i = lift (digit i)
```

```
readResult' :: CalcTransformerT Float
readResult' = lift readResult
```

Taschenrechner (3)

`total` und `clear` anpassen, damit diese `IO` durchführen:

```
total' =
  STT $ \(fn,zahl) -> do
    let res = ((), (id, fn zahl))
    putStr $ show (fn zahl)
    return res

clear' =
  STT $ \(fn,zahl) ->
    if zahl == 0.0 then do
      putStr ("\r" ++ (replicate 100 ' ') ++ "\r")
      return ((),startState)
    else do
      let l = length (show zahl)
      putStr $ (replicate l '\b') ++ (replicate l ' ')
              ++ (replicate l '\b') ++ "\n"

      return ((),(fn,0.0))
```

Taschenrechner (4)

`calcStep` analog wie vorher:

```
calcStep' :: Char -> CalcTransformerT ()
calcStep' x
  | isDigit x = digit' (fromIntegral $ digitToInt x)

calcStep' '+' = oper' (+)
calcStep' '-' = oper' (-)
calcStep' '*' = oper' (*)
calcStep' '/' = oper' (/)
calcStep' '=' = total'
calcStep' 'c' = clear'
calcStep' _   = STT $ \(fn,z) -> return ((),(fn,z))
```

Taschenrechner (5)

Für die Hauptschleife müssen [IO-Funktionen](#) in den `CalcTransformerT` geliftet werden:

```
liftIO :: IO a -> CalcTransformerT a
liftIO akt = STT (\s -> do
    r <- akt
    return (r,s))

calc' :: CalcTransformerT ()
calc' = do c <- liftIO $ getChar
    if c /= '\n' then do
        calcStep' c
        calc'
    else return ()
```

Taschenrechner (6)

Hauptprogramm:

```
main = do
    hSetBuffering stdin NoBuffering
    hSetBuffering stdout NoBuffering
    runST' $ calc'

runST' (STT s) = s startState
```