Programmierung in Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 30. Dezember 2020

Übersicht

Funktionale Programmiersprachen

Ideen der funktionalen Programmiersprachen:

- Programme bestehen nur aus Funktionen (keine Prozeduren, Methoden, usw.)
- Ersetze imperative Konstrukte (vor allem Schleifen) durch Rekursion
- Bei puren funktionalen Programmiersprachen: Funktionen sind im engen mathematischen Sinn zu sehen, als Abbildung von Werten auf Werte

Funktionale Programmiersprachen (2)

- Seiteneffekte verboten: Funktionen angewendet auf Argumente produzieren einen Wert, dabei gibt es keine (sichtbare) Speicheränderungen oder ähnlich
- Referentielle Transparenz: Gleiche Werte angewendet auf gleiche Funktion, liefert immer das gleiche Ergebnis

Funktionale Programmiersprachen (2)

- Seiteneffekte verboten: Funktionen angewendet auf Argumente produzieren einen Wert, dabei gibt es keine (sichtbare) Speicheränderungen oder ähnlich
- Referentielle Transparenz: Gleiche Werte angewendet auf gleiche Funktion, liefert immer das gleiche Ergebnis
- insbesondere keine Zuweisung: Variablen in funktionalen Programmiersprachen stehen für feste Werte, nicht für veränderliche Objekte

Funktionale Programmiersprachen (3)

- Programmieren = Schreiben von Funktionen
- Ausführung der Programme:
  imperative Sprachen: Abarbeiten einer Befehlsfolge
  bei funktionalen Sprachen: Auswertung von Ausdrücken
- Resultat: Wert des Ausdrucks

Funktionale Programmiersprachen (3)

- Programmieren = Schreiben von Funktionen
- Ausführung der Programme:
  imperative Sprachen: Abarbeiten einer Befehlsfolge
  bei funktionalen Sprachen: Auswertung von Ausdrücken
- Resultat: Wert des Ausdrucks

Beispiel: Berechne Quadratzahl

- Programmieren: `quadrat x = x * x`
- Ausführen: Werte `quadrat 5` aus

$$\texttt{quadrat}\ 5 \rightarrow 5 * 5 \rightarrow 25$$

- Resultat: 25

Funktionale Programmiersprachen (3)

- Modulare Programmierung: Einzelne Funktionalitäten in Funktionen auslagern
- Erstellen von größeren Funktionalitäten durch Komposition von Funktionen
- Beispiel:

```
quadriereUndVerdopple x = verdopple (quadrat x)
```

äquivalent dazu:

```
quadriereUndVerdopple = verdopple . quadrat
```

Haskell – Ein wenig Syntax

- Funktionsargumente werden “curried” geschrieben: Statt

  ```
  f(x,y) = x*y
  ```

  schreibt man in Haskell

  ```
  f x y = x * y
  ```

- Pattern-matching: Definition von Funktionen fallweise, Abarbeitung von oben nach unten, z.B.

  ```
  fakultaet 0  = 1
  fakultaet x  = x*(fakultaet (x-1))
  ```

Haskell – Ein wenig Syntax (2)

Anonyme-Funktionen: Lambda-Notation

Beispiele:

- `\x -> x` entspricht der Identitätsfunktion (`id x = x`)
- `\x y -> x` entspricht der const-Funktion (`const x y = x`)
- `\x -> x*x` entspricht der quadrat-Funktion

Vorteil:

- Man kann Funktionen "in-place" definieren und verwenden
- Bsp.: `quadriereUndVerdopple = (\x -> x+x) . (\y -> y*y)`

Haskell – Ein wenig Syntax (3)

Einige primitive und vordefinierte Datentypen

- Zahlen: `Int`, `Integer` (beliebig groß), `Float`, `Double`, etc.
- Boolesche Werte: `True`, `False`
- Tupel: `(True,5,False)` usw.

Einige Konstrukte:

- `if` Ausdruck `then` Ausdruck `else` Ausdruck
- `let` $f_1\ x_{1,1}\ \ldots\ x_{1,m} = \text{Expr}_1$
  $\ldots$
  $f_n\ x_{n,1}\ \ldots\ x_{n,m} = \text{Expr}_n$
  `in` Expr
- Nachgestelltes `let` (ungefähr):
  `where` $f_1\ x_{1,1}\ \ldots\ x_{1,m} = \text{Expr}$
- Operationen: `+`, `-`, `*`, `/`, `==`, `<=`, `>=`, `<`,`>`, `/=`

Auswertung in Haskell

Haskell wertet verzögert (nicht-strikt, lazy, call-by-need) aus:

Funktionsanwendung:
Argumente werden vor der Einsetzung nicht ausgewertet

Prinzip:
Werte erst aus, wenn Wert benötigt wird

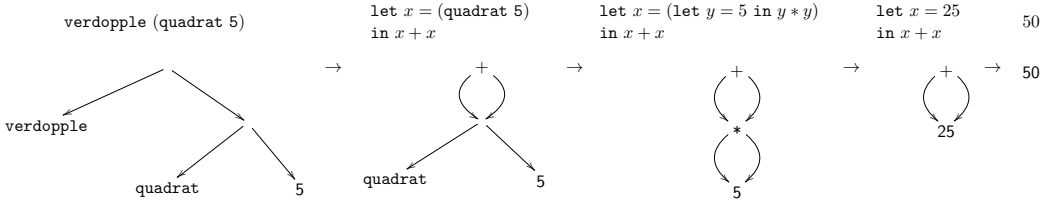
Beispiel:

```
verdopple x = x + x
quadrat x = x * x
```

nicht-strikte Auswertung (Haskell)	strikte Auswertung
`verdopple (quadrat 5)`	`verdopple (quadrat 5)`
$\rightarrow$ `(quadrat 5) + (quadrat 5)`	$\rightarrow$ `verdopple (5 * 5)`
$\rightarrow$ `(5 * 5) + (quadrat 5)`	$\rightarrow$ `verdopple 25`
$\rightarrow$ `25 + (quadrat 5)`	$\rightarrow$ `25 + 25`
$\rightarrow$ `25 + (5 * 5)`	$\rightarrow$ `50`
$\rightarrow$ `25 + 25`	
$\rightarrow$ `50`	

Auswertung in Haskell (2)

In Realität: Keine Einsetzung, sondern sharing:



Auswertung in Haskell (3)

Vorteil der call-by-need Auswertung:

- Aufrufe terminieren häufiger

  $$\texttt{const x y} = \texttt{x}$$
  $$\texttt{loop x} = \texttt{x} + \texttt{loop x}$$

  Call-by-need Auswertung:

  $\texttt{const}\ 5\ (\texttt{loop}\ 3) \to 5$

  Strikte Auswertung:

  $\texttt{const}\ 5\ (\texttt{loop}\ 3) \to \texttt{const}\ 5\ (3 + \texttt{loop}\ 3) \to \texttt{const}\ 5(3 + (3 + \texttt{loop}\ 3)) \to \ldots$

- Umgang mit unendlichen Datenstrukturen funktioniert (später)

Higher-Order Functions

- Funktionen sind ganz normale Werte
- Funktionen können Parameter anderer Funktion sein
- Funktionen können Ergebnisse von Funktionsanwendungen sein

Beispiele

- Umdrehen der Argumente einer Funktion:

```
flip f a b = f b a
```

- Funktionsionskomposition:

```
(.) f g x = f (g x)
```

Typsystem von Haskell

Haskell ist stark, statisch, polymorph getypt

- stark: jeder Ausdruck hat einen Typ
- statisch: Zur Compilezeit kann Typcheck durchgeführt werden, keine Typfehler zur Laufzeit
- polymorph: Typen können Typvariablen enthalten (Funktionen können einen schematischen Typ haben)
- Typen können vom Compiler hergeleitet werden, müssen also nicht angegeben werden

Beispiele:

```
quadrat :: Int -> Int
const :: a -> b -> a
(.) :: (b -> c) -> (a -> b) -> a -> c
flip :: (a -> b -> c) -> b -> a -> c
```

Rekursive Datenstrukturen: Listen

Listen werden konstruiert mit

- `[]` = "Nil" = leere Liste
- `:` = 'Cons', Listenkonstruktor

  Typ von Cons: `(:) :: a -> [a] -> [a]`
- Syntaktischer Zucker: [1,2,3] = 1:(2:(3:[]))
- Pattern-matching wird benutzt um Listen zu zerlegen.

Beispiele:

```
tail []     = []
tail (x:xs) = xs

head [] = error "empty list"
head (x:xs) = x
```

Listen: Beispiel

```
map f [] = []
map f (x:xs) = (f x):map f xs

filter p [] = []
filter p (x:xs) = if p x then
                   x:(filter p xs)
                  else (filter p xs)
```

Beispielverwendungen

```
map (*4) [1,2,3,4] ergibt [4,8,12,16]

filter even [1,2,3,4,5] ergibt [2,4]
```

Unendliche Listen

```
repeat x = x:(repeat x)
```

`repeat 1 ergibt [1,1,1,1,1,1,.....`

Aber (wegen nicht-strikter Auswertung)

`head (repeat 1)` wertet aus zu `1`

Noch ein paar Listenfunktionen

```
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)

reverse []     = []
reverse (x:xs) = (reverse xs) ++ x

concat []       = []
concat (xs:xxs) = xs ++ concat xxs

take 0 _        = []
take i []       = []
take i (x:xs) = x:(take (i-1) xs)
```

List comprehensions

Mengenausdrücke, die Listen beschreiben (generieren)

- $\{x^2 \mid x \in \mathbb{N}, x > 5, \text{gerade } x\}$

  als List comprehension

  ```
  [x^2 | x <- [1..], x > 4, even x]
  ```

  ```
  Test: take 10 [x^2 | x <- [1..], x > 4, even x]
  [36,64,100,144,196,256,324,400,484,576]
  ```

- $\{(x, y) \mid x \in \{1, 2, 3, 4, 5\}, y \in \{A, B, C\}\}$

  als List comprehension

  ```
  [(x,y) | x <- [1,2,3,4,5], y <- ['A','B','C']]
  ```

  ergibt

  ```
  [(1,'A'),(1,'B'),(1,'C'),(2,'A'),(2,'B'),(2,'C'),(3,'A'),
   (3,'B'),(3,'C'),(4,'A'),(4,'B'),(4,'C'),(5,'A'),(5,'B'),(5,'C')]
  ```

Selbst definierte Datentypen

Aufzählungstypen

```
data Bool       = True | False

data RGB        = Rot | Gruen | Blau
```

Pattern-Matching zum Selektieren

```
zeigeAn :: RGB -> String
zeigeAn Rot   = "Farbe Rot"
zeigeAn Gruen = "Farbe Gr\"un"
zeigeAn Blau  = "Farbe Blau"

isGruen :: RGB -> Bool
istGruen Gruen = True
istGruen _     = False
```

Selbst definierte Datentypen (2)

Polymorphe Datentypen

```
data Maybe a    = Nothing | Just a

data Either a b = Left a | Right b

lookup :: a -> [(a,b)] -> Maybe b
lookup x []         = Nothing
lookup x ((y,z):ys) =
  if x == y then Just z
  else lookup x ys
```

Selbst definierte Datentypen (3)

Rekursive Datentypen

```
data IntBaum    = Blatt Int | Knoten Int IntBaum Intbaum

data PolyBaum a = Blatt a
                | Knoten a (PolyBaum a) (PolyBaum a)

data List a     = Nil | Cons a (List a)
```

Beispiel:

```
mapBaum :: (a -> b) -> PolyBaum a -> PolyBaum b
mapBaum f (Blatt x)              = Blatt (f x)
mapBaum f (Knoten x bauml baumr) =
 Knoten (f x) (mapBaum f bauml) (mapBaum f baumr)
```

Selbst definierte Datentypen (4)

Typsynonyme

```
type Koordinate = (Int,Int)
type IntBaum    = PolyBaum Int
type MeineListe a = [a]
```

Typklassen

- Zur Überladung von Operatoren, z.B. `(+)`, `(==)` für verschiedene Typen
- Beispiele

```
(+) :: (Num a) => a -> a -> a

lookup :: (Eq a) => a -> [(a, b)] -> Maybe b

show :: (Show a) => a -> String
```

Ein- und Ausgabe in Haskell

Problematik

Echte Seiteneffekte sind in Haskell nicht erlaubt.

Warum?

Problematik

Echte Seiteneffekte sind in Haskell nicht erlaubt.

Warum?

Annahme: `getZahl` wäre eine "Funktion", die eine Zahl von der Standardeingabe liest

Problematik

Echte Seiteneffekte sind in Haskell nicht erlaubt.

Warum?

Annahme: `getZahl` wäre eine "Funktion", die eine Zahl von der Standardeingabe liest

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

Problematik

Echte Seiteneffekte sind in Haskell nicht erlaubt.

Warum?

Annahme: `getZahl` wäre eine "Funktion", die eine Zahl von der Standardeingabe liest

- Referentielle Transparenz ist verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.
- Gelten noch mathematische Gleichheiten wie $e + e = 2 * e$?

Problematik

Echte Seiteneffekte sind in Haskell nicht erlaubt.

Warum?

Annahme: `getZahl` wäre eine "Funktion", die eine Zahl von der Standardeingabe liest

- Referentielle Transparenz ist verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.
- Gelten noch mathematische Gleichheiten wie $e + e = 2 * e$?
  Nicht für $e =$ `getZahl`!

Problematik

Echte Seiteneffekte sind in Haskell nicht erlaubt.

Warum?

Annahme: `getZahl` wäre eine "Funktion", die eine Zahl von der Standardeingabe liest

- Referentielle Transparenz ist verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.
- Gelten noch mathematische Gleichheiten wie $e + e = 2 * e$?
  Nicht für $e =$ `getZahl`!
- `length`$[$`getZahl`$,$ `getZahl`$]$ fragt nach gar keiner Zahl?

```
length [] = 0
length (_:xs) = 1 + length xs
```

Problematik

Echte Seiteneffekte sind in Haskell nicht erlaubt.

Warum?

Annahme: `getZahl` wäre eine "Funktion", die eine Zahl von der Standardeingabe liest

- Referentielle Transparenz ist verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.
- Gelten noch mathematische Gleichheiten wie $e + e = 2 * e$?
  Nicht für $e =$ `getZahl`!
- `length`$[$`getZahl`$,$ `getZahl`$]$ fragt nach gar keiner Zahl?

```
length [] = 0
length (_:xs) = 1 + length xs
```

- Festlegung auf eine genaue Auswertungsreihenfolge nötig. (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-/Ausgabe durchführt und anschließend einen Wert vom Typ `a` liefert.

Monadisches I/O

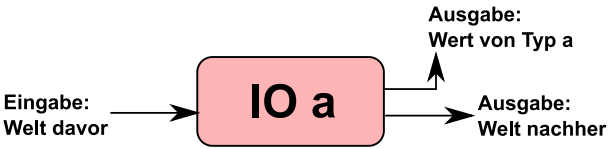
Kapselung des I/O

- Datentyp `IO a`
- Wert des Typs `IO a` ist eine I/O-Aktion, die beim Ausführen Ein-/Ausgabe durchführt und anschließend einen Wert vom Typ a liefert.
- Programmiere in Haskell I/O-Aktionen, Ausführung quasi außerhalb von Haskell

Monadisches I/O (2)

Vorstellung:

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

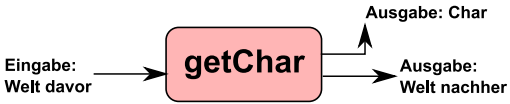


Monadisches I/O (3)

- Werte vom Typ `IO a` sind Werte, d.h. sie können nicht weiter ausgewertet werden
- Sie können allerdings ausgeführt werden (als Aktion)
- In der Vorstellung: erst dann wenn eine Welt reingesteckt wird

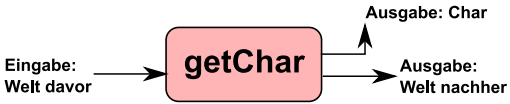
Primitive I/O-Operationen

```
getChar :: IO Char
```

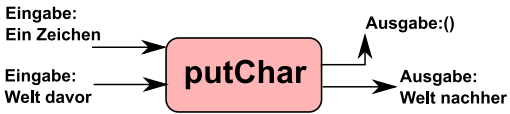


Primitive I/O-Operationen

```
getChar :: IO Char
```

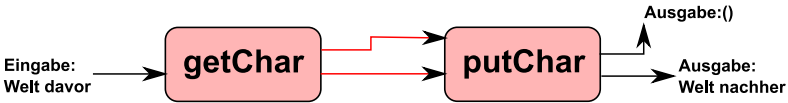



```
putChar :: Char -> IO ()
```



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`)

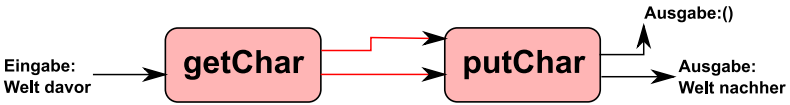


I/O Aktionen komponieren

Der >>= Operator (gesprochen: “bind”)

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

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



I/O Aktionen komponieren (3)

Der >> Operator (gesprochen: “then”)

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

Kann mit >>= definiert werden:

```
(>>) :: IO a -> IO b -> IO b
(>>) akt1 akt2 = akt1 >>= \_ -> akt2
```

I/O Aktionen komponieren (4)

Gelesenes Zeichen zweimal ausdrucken

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

I/O Aktionen komponieren (5)

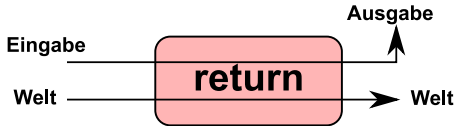
Neue I/O-Aktionen bauen, die zwei Zeichen liest und als Paar zurück liefert

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= \x ->
              getChar >>= \y ->
              ???
```

I/O Aktionen komponieren (5)

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

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= \x ->
              getChar >>= \y ->
              return (x,y)
```



I/O Aktionen komponieren (6)

Syntaktischer Zucker: **do**-Notation:

```
getTwoChars :: IO (Char,Char)
getTwoChars =
              getChar >>= \x ->
              getChar >>= \y ->
              return (x,y)
```

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

do-Notation weg transformieren

do-Notation kann in >>= übersetzt werden:

```
do { x<-e; s }  = e >>= \x -> do { s }
do { e; s }     = e >> do { s }
do { e }        = e
```

Ausreichend

>>= und `return`

Wir lassen in der do-Notation, die geschweiften Klammern und ; weg.
(Der Parser fügt sie automatisch, bei richtiger Einrückung ein)

Beispiel

Eine Zeile einlesen

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

Monaden

Eine Monade besteht aus einem Typkonstruktor `M` und zwei Operationen:

```
(>>=)  :: M a -> (a -> M b) -> M b
return :: a -> M a
```

wobei zusätzlich 3 Gesetze gelten müssen.

Monadengesetze

(1) `return x >>= f` $=$ `f x`

(2) `m >>= return` $=$ `m`

(3)
```
m1 >>= (\x -> m2 >>= (\y -> m3))
```
$=$
```
(m1 >>= (\x -> m2)) >>= (\y -> m3)
```

- Man kann nachweisen, dass `M` $=$ `IO` mit `>>=` und `return` eine Monade ist.
- Monade erzwingt Sequentalisierung ("I/O ist richtig implementiert")

Monadisches I/O: Anmerkungen

- Beachte: Es gibt keinen Weg aus der IO-Monade heraus
- Aus I/O-Aktionen können nur I/O-Aktionen zusammengesetzt werden
- Keine Funktion vom Typ `IO a -> a`!

Monadisches I/O: Anmerkungen

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

Beispiel: readFile

`readFile`: Liest den Dateiinhalt 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)
```

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.

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)
```

Handle auslesen: verzögert

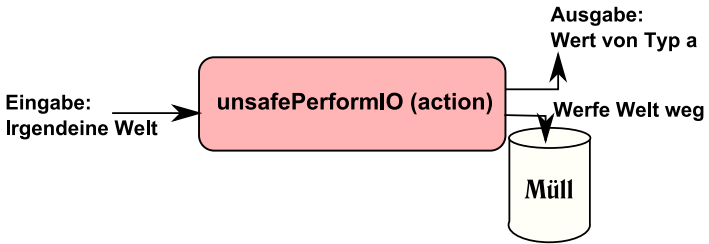
```
leseHandleAus handle =
 do
  ende <- hIsEOF handle
  if ende then hClose handle >> return []
   else
    do
     c  <- hGetChar handle
     cs <- unsafeInterleaveIO (leseHandleAus handle)
     return (c:cs)
```

Test:

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

UnsafePerformIO

- `unsafePerformIO :: IO a -> a`
- “knackt” die Monade



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

Nützliche Kombinatoren für monadisches IO

(Die Typen der Funktionen in aktuellem Haskell sind allgemeiner)

```
sequence :: [IO a] -> IO [a]
```

führt Liste von IO-Aktionen nacheinander aus, liefert Liste der Ergebnisse

```
sequence_ :: [IO a] -> IO ()
```

führt Liste von IO-Aktionen nacheinander aus, verwirft Ergebnisse

```
mapM :: (a -> IO b) -> [a] -> IO [b]

mapM f xs = sequence (map f xs)

mapM_ :: (a -> IO b) -> [a] -> IO ()

mapM_ f xs = sequence_ (map f xs)
```

Veränderliche Speicherplätze

```
data IORef a -- Abstrakter Typ

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

Veränderliche Speicherplätze

```
data IORef a -- Abstrakter Typ

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

Imperativ (z.B. C):

```
int x := 0
x := x+1
```

Veränderliche Speicherplätze

```
data IORef a -- Abstrakter Typ

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

Imperativ (z.B. C):

```
int x := 0
x := x+1
```

In Haskell mit IORefs

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