Kapitel 10: Andere funktionale Sprachen: Haskell

Andreas Abel

LFE Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München

1. Juli 2011

Inhalt

Haskell

- Effektfreie funktionale Sprache mit verzögerter Auswertung.
- Benannt nach *Haskell Curry* (1900-82).
- Standards: **Haskell 98** und **Haskell 2010**.
- Implementierungen: Hugs, NHC, ..., **GHC**.
- **G**lorious **H**askell **C**ompiler entwickelt zuerst an der Uni Glasgow und nun bei Microsoft Research Cambridge.
- Die Hauptentwickler Simon Peyton-Jones und Simon Marlow bekamen 2011 den ACM SIGPLAN *Programming Language Software Award*.

Haskell-Syntax

- Wenig Schlüsselwörter.
- Wenig Klammerung.
- **Einrückung** ist Teil der Syntax! (wie by Python)
- Keine Wertzuweisung (:=).
- Benutzerdefinierte **Infixoperatoren**.
- Benutzerdefinierte **überladene Operationen** mittels **Typklassen**.
- Simples hierarchisches Modulsystem; Modulnamen entsprechen Dateinamen.

SML vs. Haskell-Syntax: Funktionen

SML-Syntax

```
(* SML comment block *)


(* function abstraction *)
fn x => fn y => x
fn (x,y) => y
fn []      => true
 | (x::xs) => false

(* function declaration *)
fun iter f a 0 = a
  | iter f a n =
      iter f (f a) (n-1)
```

Haskell-Syntax

```
{- Haskell comment block -}
-- Haskell one line comment


-- \ for λ
\ x y   -> x
\ (x,y) -> y
{- no multi-clause anonymous
   functions -}

-- just equations
iter f a 0 = a
iter f a n =
  iter f (f a) (n-1)
```

SML vs. Haskell-Syntax: Listen

Rollentausch: In Haskell ist `:` Listenkonstruktion (SML `::`) und `::` Typzuweisung (SML `:`).

```
[]        (* empty list  *)        []
x::xs     (* cons        *)        x:xs
[1,2,3]   (* literal     *)        [1,2,3] {- or -} [1..3]
l @ l'    (* append      *)        l ++ l'
```

Typsignaturen `bezeichner :: typ` sind optional.

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

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) =
  f x : map f xs
```

SML vs. Haskell-Syntax: Datentypen

Datenkonstruktoren sind in Haskell ge**curry**te Funktionen.

```
datatype 'a tree
  = Leaf of 'a
  | Node of 'a tree * 'a tree

(* Leaf : 'a -> 'a tree *)
fun node l r = Node (l, r)


(* case distinction *)
case t
  of Leaf(a)   => [a]
   | Node(l,r) => f l @ f r
```

```
data Tree a
  = Leaf a
  | Node (Tree a) (Tree a)

-- Leaf :: a -> Tree a
{- Node :: Tree a -> Tree a ->
     Tree a -}


-- similar to SML
case t of
  Leaf a   -> [a]
  Node l r -> f l ++ f r
```

Datentypen und Konstruktoren müssen in Haskell **Groß** geschrieben werden. In SML hat Großschreibung keine lexikalische Signifikanz.

SML vs. Haskell-Syntax: Lokale Definitionen

In Haskell können Definitionen mit `where` nachgestellt werden.

```
let val k     = 5
    fun f 0   = k
      | f n   = f (n-1) * n
in  f k
end
```

```
let k   = 5
    f 0 = k
    f n = f (n-1) * n
in  f k
```

```
(* no post-definition *)
```

```
f k where
      f 0 = k
      f n = f (n-1) * n
      k   = 5
```

```
local ... in ... end
```

```
-- nothing corresponding
```

SML bearbeitet Definitionsfolgen von vorne nach hinten. In Haskell ist die Reihenfolge der Definitionen unerheblich.

SML vs. Haskell-Syntax: Rekursion

In Haskell sind alle Gleichungen wechselseitig rekursiv.

```
(* mutual recursion *)
fun even 0 = true
  | even n = odd (n-1)
and odd  0 = false
  | odd  n = even (n-1)

val rec f = fn x =>
  if x <= 1 then 1
    else x * f (x-1)
```

```
-- no need to mark mutual rec.
even 0 = true
even n = odd (n-1)
odd  0 = false
odd  n = even (n-1)

f = \ x ->
  if x <= 1 then 1
    else x * f (x-1)
```

In SML benötigt Rekursion das Schlüsselwort **rec** oder **fun**...**and**....

SML vs. Haskell-Syntax: Module

Haskell hat ein simples Modulsystem (keine Signaturen, keine Funktoren).

```
structure M = struct
  ...
end
```

```
module M where
...
```

```
open Text.Pretty
structure S = System.IO
```

```
import Text.Pretty
import qualified System.IO as S
```

Auswertungsstrategie

- Eine **Auswertungsstrategie** beschreibt, wie ein Ausdruck ausgewertet wird. Insbesondere:
  1. In welcher Reihenfolge werden die Teilausdrücke bearbeitet?
  2. Werden Funktionsargumente ausgewertet, bevor die Funktion aufgerufen wird?
- **Effektfreie** bzw. **referenziell transparente** Programme verhalten sich unter allen Auswertungsstrategien gleich.
- Bei Effekten (Wertzuweisung, Ausnahmen, Ein/Ausgabe) ist die Auswertungsstrategie signifikant.
- Welche Ausgabe (mittels `print`) erzeugt folgendes Programm?

```
fun f1 x y z = y + y + z

val t1 = f1 (print "first\n"; 1)
            (print "second\n"; 2)
            (print "third\n"; 3)
```

Applikative Auswertung (call-by-value)

- In SML (wie fast allen Sprachen) wird **applikativ** und **von links nach rechts** ausgewertet:
  1. Vor dem Funktionsaufruf werden die Argumente ausgewertet **(call-by-value)**.
  2. Anweisungen und Argument werden von links nach rechts, oben nach unten ausgewertet.
- Vor dem Aufruf von `f1` wird zuerst das erste Argument `(print "first\n"; 1)` bearbeitet. Der Effekt ist das Drucken von `first`, das Ergebnis `1`.
- Dann zweites und drittes Argument, dann der Aufruf `f1 1 2 3`.

```
first
second
third
val t1 = 7 : int
```

Normale Auswertung (call-by-name)

- Bei der normalen Auswertung werden die Argumente unausgewertet in den Funktionsrumpf eingesetzt **(call-by-name)**.
- Beispiel: **var**-Parameter in **PASCAL**.

```
program SquareFive;
  procedure times (n : integer; var x : integer);
  begin
    x := n * x;
  end;
var i : integer;
begin
  i := 5;
  times (i, i);
  println (i);  (* Prints 25 *)
end.
```

- Als erstes Argument wird der **Wert** von `i` übergeben (call-by-value), als zweites der **Name** `i` (call-by-name), der dann neu zugewiesen wird.

Simulation von *call-by-name* durch *thunks*

- Was wäre die Ausgabe von `t1`, hätte SML normale Auswertung?

```
fun f1 x y z = y + y + z

val t1 = f1 (print "first\n"; 1)
            (print "second\n"; 2)
            (print "third\n"; 3)
```

- Wir simulieren call-by-name durch **thunks** `fn () => e`.

```
fun f2 x y z = y() + y() + z()

val t2 = f2 (fn () => (print "first\n"; 1))
            (fn () => (print "second\n"; 2))
            (fn () => (print "third\n"; 3))
```

- Eine Funktion `fn ... => ...` ist bereits ein Wert.

Ineffizienz von call-by-name

- Die Ausgabe von `t2` ist nun:

```
second
second
third
val f2 = 7 : int
```

- Argument 1 wird nicht ausgewertet, Argument 2 zweimal.
- Der Effekt von Argument 2 tritt zweimal auf.
- *call-by-name* ist ineffizient für Funktionen, die ihr Argument mehrfach benutzen.

```
fun bla x =
  if x = 0 then ...
  else if x = 1 then ...
  else if x = 2 then ...
  else ...
```

Verzögerte Auswertung (call-by-need)

- Vorteil von call-by-value: Jedes Argument wird nur 1x ausgewertet.
- Vorteil von call-by-name: Unbenutzte Argumente werden gar nicht ausgewertet.
- **Verzögerte Auswertung (call-by-need)** kombiniert diese Vorteile:
  1. Ausdrücke werden nur ausgewertet, wenn ihr Wert **benötigt** wird.
  2. Der errechnete Wert wird gespeichert (memoisiert) für mehrmalige Verwendung.
- Auch **lazy evaluation** (faule Auswertung) genannt.
- Welche Ausgabe produziert die verzögerte Auswertung von `t1`?

```
fun f1 x y z = y + y + z

val t1 = f1 (print "first\n"; 1)
            (print "second\n"; 2)
            (print "third\n"; 3)
```

Memoizing thunks

- `memo` erzeugt aus einem Thunk `f` eine memoisierende Version.

```
fun memo (f : unit -> 'a) : unit -> 'a =
  let val r = ref NONE
  in  fn () => case !r
         of NONE =>
              let val v = f()
              in  r := SOME v;
                  v
              end
          | SOME v => v
  end
```

- Die Referenz `r` hat zuerst Inhalt `NONE`.
- Wird der Wert `v` von `f` zum ersten Mal angefordert (Fall `!r = NONE`), wird er in `r` gespeichert.
- Bei folgenden Anforderungen (Fall `!r = SOME v`) wird der gespeicherte Wert `v` direkt zurückgegeben.

Simulation von verzögerter Auswertung in SML

- Mit memoisierten Thunks ...

```
fun f2 x y z = y() + y() + z()

val t3 = f2 (memo (fn () => (print "first\n"; 1)))
            (memo (fn () => (print "second\n"; 2)))
            (memo (fn () => (print "third\n"; 3)))
```

- ... erhalten wir nun diese Ausgabe:

```
second
third
val t3 = 7 : int
```

Auswertung in Haskell

- Haskell bedient sich der verzögerten Auswertung.
- Haskell ist eine **effektfreie** Sprache, auch **reine** Sprache (engl. *pure language*), jedoch:
  1. Eine nicht-fangbare Ausnahme kann mit **`error`** `"message"` geworfen werden.
  2. Logging kann mit `trace "message"` erfolgen.
- Damit können wir das Auswertungsverhalten studieren:

```
import Debug.Trace

f x y z = y + y + z
t = f (trace "first" 1) (trace "second" 2) (trace "third" 3)
```

- Rufen wir im Interpreter `t` auf, erhalten wir:

```
second
third
7
```

Verzögerte Auswertung und Sonderformen

- **Sonderformen** sind Sprachkonstrukte, die einer anderen Auswertungsstrategie folgen.
- Beispiel: `if c then t else e`
  1. Zuerst wird die Bedingung `c` ausgewertet.
  2. Ist das Ergebnis `true`, wird der Wert von `t` zurückgegeben.
  3. Ist das Ergebnis `false`, wird der Wert von `e` zurückgegeben.
- Ein naives selbst-definiertes if-then-else in SML:

```
fun ifThenElse true  t e = t
  | ifThenElse false t e = e

val t = ifThenElse (1>0) (print "wahr\n") (print "falsch\n")
```

- Was ist die Ausgabe dieses Programmstücks?

Sonderform if-then-else

- Ausgabe von

```
val t = ifThenElse (1>0) (print "wahr\n") (print "falsch\n")

wahr
falsch
val t = () : unit
```

- Problem: Beide Zweige (then und else) werden ausgewertet.
- In SML kann man if-then-else nur mit thunks definieren:

```
fun ifThenElse true  t e = t()
  | ifThenElse false t e = e()

val t = ifThenElse (1>0)
        (fun () => print "wahr\n")
        (fun () => print "falsch\n")
```

Sonderformen in Haskell

- In Haskell ist if-then-else dank verzögerter Auswertung keine Sonderform.

```
ifThenElse True t e = t
ifThenElse False t e = e

t1 = ifThenElse (1>0) (trace "wahr" ()) (trace "falsch" ())
```

- Auswertung von `t1`:

```
wahr
()
```

- Haskell ermöglicht die Definition eigener Kontrollstrukturen und **eingebetteter Sprachen** (engl. **domain specific languages**).
- Zum Spass hat Lennart Augustsson **BASIC** in Haskell eingebettet.

Anfängerfallen der verzögerten Auswertung

- Ausnahmen werden nicht unbedingt geworfen.

```
g x y = let z = x `div` y
        in  if x > y * y then x else z

t2 = g 5 0
```

- `t2` liefert `5`. Da der Wert von `z` nicht benötigt wird, kommt es nicht zur Division durch Null.
- **Sequentialität** der Auswertung (erst das, dann das) nicht gegeben.
- Das Beispiel zeigt, dass **beliebige Teilausdrücke** (hier `x `div` y`) durch eine **`let`**-gebundene Variable (hier `z`) abstrahiert werden können.
- In SML stehen Variablen für **Werte**. Nur Werte können immer abstrahiert werden, ohne das Programmverhalten zu ändern.

Potentiell unendliche Strukturen

- Haskell-Listen können unendliche Länge haben.
- Der Aufruf `iterate f a` erzeugt die Folge
  `[a, f(a), f(f(a)), f(f(f(a))), ...]`.

```
iterate :: (a -> a) -> a -> [a]
iterate f a = a : iterate f (f a)

powers2 = iterate (\ x -> x+x) 1   -- list of all powers of 2
t3 = take 10 powers2               -- take the first 10
```

- `t3` ergibt `[1,2,4,8,16,32,64,128,256,512]`.
- Die Liste `powers2` wird nie ganz erzeugt, nur soweit wie nötig.

Wertrekursion

- Bisher hatten wir nur rekursiv definierte *Funktionen*; in einer **strikten Sprache** (d.h. call-by-value) wie SML geht auch nichts anderes.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs)  = f a b : zipWith f as bs
zipWith f _      _       = []
```

- `zipWith f [a1,a2,...] [b1,b2,...] = [f a1 b1, f a2 b2, ...]`
- In einer **faulen** Sprache (d.h. call-by-name/need) wie Haskell gibt es auch rekursive **Werte**.
- Einschlägiges Beispiel ist die Sequenz der Fibonaccizahlen:

```
fib = 0 : 1 : zipWith (+) fib (tail fib)
f10 = take 10 fib
```

```
[0,1,1,2,3,5,8,13,21,34]
```

- Implementiert $\mathrm{fib}_0 = 0$ sowie $\mathrm{fib}_1 = 1$ und $\mathrm{fib}_{n+2} = \mathrm{fib}_{n+1} + \mathrm{fib}_n$.

Wächter in Fallunterscheidungen

- In Fäelle in Funktionsdefinition oder **case** können durch **Wächter** (engl. *guards*) "geschützt" werden.
- Z.B. Definition von **filter** `p l`, das alle Elemente aus Liste `l` zurückgibt, die Prädikat `p` erfüllen.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                    = []
filter p (a:as) | p a          = a : filter p as
                | otherwise = filter p as
```

- Nach den Mustern (wie `(a:as)`) können nach dem Strich `|` noch Wächter, Bool'sche Ausdrücke stehen.
- Nur wenn diese zu **True** auswerten, wird der Zweig gewählt, ansonsten zum Nächsten gegangen.
- Wächter **otherwise** ist nur ein Alias für **True**, also immer passierbar.

Erweiterte Wächter

- Neben Bool'schen Wächter gibt es noch die **pattern guards**.
- Folgende Funktion `allJust` nimmt eine Liste von optionalen Werten **`Maybe`** a und prüft, ob alle Elemente definiert sind (**`Just`** a).
  1. Ist dies der Fall, wird eine Liste der Elemente ohne **`Just`**-Konstruktor zurückgegeben.
  2. Andernfalls **`Nothing`**.

```
data Maybe a = Just a | Nothing

allJust :: [Maybe a] -> Maybe [a]
allJust []                                    = Just []
allJust (Just a : l) | Just l' <- allJust l   = Just (a : l')
allJust _                                     = Nothing
```

- Der Wächter **`Just`** `l'` `<-` `allJust l` prüft, ob das Ergebnis der Rekursion ein **`Just`** ist und bindet den Inhalt an Variable `l'`.

Nochmal **pattern guards**

- Berechnet eine Funktion ein Tupel, können wir das Ergebnis des rekursiven Aufrufs mit einem *pattern guard* zerlegen.
- Folgende Funktion **span** `p l` teilt die Liste `l` an der Stelle, wo das Prädikat `p` zum ersten Mal *nicht mehr* gilt.

```
span :: (a -> Bool) -> [a] -> ([a], [a])
span p []                          = ([], [])
span p (x : xs)
  | p x, (ys, zs) <- span p xs     = (x : ys, zs)
  | otherwise                      = ([], x : xs)
```

- Hier findet sich ein Bool'scher Wächter `p x` und ein *pattern guard* `(ys, zs) <- span p xs`.

Listenkomprehension

- In der Mathematik verwendet man **Mengenkomprehension** wie $\{x \mid x \in \mathbb{N}, x \bmod 3 = 1\}$, die Menge der natürlichen Zahlen, die bei Division durch 3 den Rest 1 haben.
- In Haskell gibt es eine analoge Notation für Listen.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p l = [ a | a <- l, p a ]
```

- Damit ist auch das kartesische Produkt zweier Listen direkt definierbar:

```
cartesian :: [a] -> [b] -> [(a,b)]
cartesian as bs = [ (a,b) | a <- as, b <- bs ]
```

- Test `cartesian [1,2] [3,4,5]` ergibt:

```
[(1,3),(1,4),(1,5),(2,3),(2,4),(2,5)]
```

Infix-, Präfix- und Sektionsschreibweise

- Einen **Infixoperator** wie + kann man in Klammern `(x)` **präfix** verwenden. Beispiel: `(+) 5 3` statt `5 + 3`.
- Dadurch ist partielle Applikation möglich: `map ((+) 3) [1..5]` (ergibt `[4..8]`).
- Alternativ dazu sind **Sektionen**: Durch Weglassen eines Operatorargumentes erhält man eine einstellige Funktion.

```
map (3 +) [1,2] == [4,5]
map (3 -) [1,2] == [2,1]
map (/ 2) [1,2] == [0.5,1.0]
```

- Jeden Bezeichner kann man in ‘*backquotes*‘ infix verwenden.

```
if 3 `elem` [1..5] then 1 else 0
```

Infixoperatoren

- Eigene Infixoperatoren muss man unter Angabe der **Bindungsstärke** deklarieren. Beispiel: Linksassoziatives Exklusiv-Oder:

```
infixl 3 /+/
(/+/)    :: Bool -> Bool -> Bool
x /+/ y  =  not (x == y)
```

- Funktionskomposition (in SML: o) hat die höchste Bindungsstärke:

```
infixr 9 .
(.)      :: (b -> c) -> (a -> b) -> (a -> c)
(f . g)  =  \ x -> f (g x)
```

- Applikation mit `$` hat die niedrigste Bindungsstärke:

```
infixr 0 $
($)      :: (a -> b) -> a -> b
f $ x    =  f x
```

- So kann man `3 + (5 + 4)` auch schreiben: `(3 +) . (+) 5 $ 4`.

Klammen sparen mit `$`

- Wozu noch einen Operator `$` für Applikation? Klammern sparen!
- Das letzte Argument einer Funktion kann man mit `$` abteilen und braucht es dann nicht Klammern.
- Statt `fun1 (fun2 arg2 (fun3 (fun4 arg4)))`

```
fun1 $ fun2 arg2 $ fun3 $ fun4 arg4
```

- Weitere Infixoperatoren:

```
infix  4  ==, /=, <, <=, >=, >   -- comparison
infixr 3  &&                     -- boolean and
infixr 2  ||                     -- boolean or
```

Prinzipale Typen überladener Operatoren

- Die Operation $+$ ist in SML **überladen** (engl. *overladed*).
- Sie auf `int` und `real` angewendet werden.

```
+ : int -> int -> int
+ : real -> real -> real
```

- Im mehrdeutingen Fall `fn x => fn y => x + y` wählt SML willkürlich `int -> int -> int`.
- Der **prinzipale Typ** von + wäre jedoch (in hypothetischer Syntax):

$$\texttt{+ : (}\alpha \in \{\texttt{int,real}\}\texttt{) => } \alpha \texttt{ -> } \alpha \texttt{ -> } \alpha$$

- Definierten wir das Prädikat "numerischer Typ" `Num` α als $\alpha \in \{\text{int,real}\}$, so könnten wir schreiben:

$$\texttt{+ : (Num } \alpha\texttt{) => } \alpha \texttt{ -> } \alpha \texttt{ -> } \alpha$$

Überladene Operatoren in Haskell

- Der Haskell-Interpreter GHCI liefert auf die Typabfrage `:type (+)`

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

- Dies besagt, dass `(+)` den Typen `a -> a -> a` hat falls `a` der Typklasse `Num` angehört.
- Die Typklasse `Num` ist folgendermassen definiert (einsehbar mit `:info Num`):

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  ...
```

- Auch Zahlkonstanten sind überladen, siehe z.B. `:type 5`.

```
5 :: Num a => a
```

Überladene Gleichheit in Haskell

- Test auf Gleichheit ist das überladene Symbol ==.
- Keine Sonderbehandlung wie in SML (Gleichheitstypen) nötig.
- Gleichheit ist eine Typklasse. Sie definiert zwei überladene Operatoren == und <=.

```
class Eq a where
    (==), (/=)      :: a -> a -> Bool
    x /= y          =  not (x == y)
    x == y          =  not (x /= y)
```

- Ungleichheit <= ist standardmäßig durch == definiert und umgekehrt.
- Eine Instanziierung muss wenigstens eine der beiden Operationen implementieren:

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

Gleichheit auf eigenen Datentypen

- Gleichheitstests für Datentypen sind schematisch:

```
data Expr = Number Int | Plus Expr Expr

instance Eq Expr where
  Number i    == Number i'     =  i == i'
  Plus e1 e2 == Plus e1' e2' =  e1 == e1' && e2 == e2'
```

- Diese Instanz kann GHC für uns berechnen.

```
data Expr = Number Int | Plus Expr Expr
            deriving Eq
```

- Im Gegensatz zu SML gibt es keine festverdrahtete Druckfunktion für benutzerdefinierte Datentypen. Beispielseingabe: `Number 3`

```
No instance for (Show Expr)
  arising from a use of ‘print’
Possible fix: add an instance declaration for (Show Expr)
```

Typklasse **Show** zum Darstellen von Werten

- Eine Standarddruckfunktion ist automatisch herleitbar:

```
data Expr = Number Int | Plus Expr Expr
          deriving (Eq, Show)
```

- Meist implementiert man jedoch eine lesbarere Form:

```
instance Show Expr where

  show (Number i)  = show i
  show (Plus e e') = parens $ show e ++ " + " ++ show e'

    where parens s = "(" ++ s ++ ")"
```

Klassen polymorpher Typen

- Naive Instanziierung von Typklassen für polymorphe Typen schlägt fehl.

```
data Maybe a = Just a | Nothing

instance Eq (Maybe a) where
  Just a   == Just a' = a == a'
  Nothing  == Nothing = True
  _        == _       = False
```

```
No instance for (Eq a)
  arising from a use of ‘==’
In the expression: a == a'
In an equation for ‘==’: (Just a) == (Just a') = a == a'
In the instance declaration for ‘Eq (Maybe a)’
```

- In der Tat, eine Funktion `(==) :: Maybe a -> Maybe a -> Bool` können wir nicht implementieren.

Typklassen-Annahmen

- Auf einem unbekannten Typen a können wir keine Gleichheit implementieren, sie muss gegeben sein.

```
eqMaybe :: (a -> a -> Bool) -> Maybe a -> Maybe a -> Bool
eqMaybe eqA (Just a)  (Just a')  = eqA a a'
eqMaybe eqA Nothing   Nothing    = True
eqMaybe eqA _         _          = False
```

- Analog können wir **Typklassenannahmen** (engl. *type class constraints*) machen.

```
instance Eq a => Eq (Maybe a) where
  ...
```

Wenn a Instanz von **Eq** ist, dann auch **Maybe** a.

Typklassen benutzen

- Die Funktion **group** 1 fasst aufeinanderfolgende gleiche Elemente der Liste ||l|| zusammen.

```
group :: Eq a => [a] -> [[a]]
group []                                    = []
group (x:xs) | (ys, zs) <- span (x ==) xs = (x : ys) : group zs

group "Mississippi" == ["M","i","ss","i","ss","i","pp","i"]
```

- Diese polymorphe Funktion kann nur für Typen a benutzt werden, die die Typklasse **Eq** instanziieren.
- Zur Laufzeit wird für **Eq** a ein **Wörterbuch** (engl. *dictionary*) übergeben, das die Implementierungen von der Klassenmethoden == und /= für den Typen a enthält.

Simulation von Typklassen in SML

- Eine Typklasse ist ein **Verbundtyp** (engl. *record type*).

```
type 'a eq =
  { eq   : 'a -> 'a -> bool (* equality   *)
  , ineq : 'a -> 'a -> bool (* inequality *)
  }
```

- Instanzen sind *Verbände* (engl. *records*) diese Typs.

```
val intEq : int eq =
  { eq   = fn x => fn y => x = y
  , ineq = fn x => fn y => x <> y
  }

val charEq : char eq =
  { eq   = fn x => fn y => x = y
  , ineq = fn x => fn y => x <> y
  }
```

Simulation von Typklassen in SML

- Eine Typklass-polymorphe Funktion erwartet eine Typklass-Parameter.

```
fun group (aEq : 'a eq) ([] : 'a list) = []
  | group (aEq : 'a eq) (x :: xs)  =
      let val (ys, zs) = span (fn y => #eq aEq x y) xs
      in  (x :: ys) :: group aEq zs
      end
```

- Die Typklass-Instanz wird dann manuell übergeben.

```
val testGroup = map String.implode
     (group charEq (String.explode "Mississippi"))
```

```
val testGroup = ["M","i","ss","i","ss","i","pp","i"] : string list
```

Simulation von Typklassen in SML

- Polymorphe Typklass-Instanzen bilden *records* auf *records* ab.

```
fun eqList (eqA : 'a eq) : 'a list eq =
  let fun eqL [] []                   =  true
        | eqL (x :: xs) (y :: ys) =  #eq eqA x y andalso
                                        eqL xs ys
        | eqL _ _                     =  false
  in { eq   = eqL
     , ineq = fn xs => fn ys => not (eqL xs ys)
     }
  end
```

Zusammenfassung Typklassen

- Überladung wird in Haskell realisiert durch Typklassen und deren Instanzen.
- Instanzen werden an Typklass-polymorphe Funktionen als implizite Parameter (dictionaries) übergeben.
- Welches *dictionary* übergeben werden muss, findet der Übersetzer durch die Typisierung heraus. (Typinferenz erledigt in diesem Fall Arbeit für den Programmierer.)
- Konsequenz: es kann nur eine Instanz pro Typ geben.
- Manchmal möchte man verschiedene Gleichheitsbegriffe wie z.B. Listenidentität oder Gleichheit modulo Permutation. Dies kann man durch einen Wrapper **newtype** realisieren.

```
newtype PermList a = PL [a]

instance Eq a => Eq (PermList a) where
  PL l == PL l' = sort l == sort l'
```

Ein-/Ausgabe

- Haskell ist eine **reine** funktionale Sprache. Gewöhnliche Ausdrücke haben *keinen* **Seiteneffekt**.
- Effekte erfordern **sequenzielle Auswertung**. Dies ist by **call-by-need** nicht gegeben.
- Sequenzialisierung kann in Haskell mittels der **do**-Notation erzwungen werden.

```
main = do
  putStrLn "Hello, world!"
  putStrLn "Bye-bye!"
```

- Der Typ von `main` ist `IO ()`.
- D.h. `main` hat kein eigentliches Ergebnis (Rückgabetyp ist der unit-Typ `()`), jedoch tätigt es **Ein-/Ausgabe** (engl. input/output).

Ein-/Ausgabe

- Die einfachste Form des Unix-Dienstprogramms `cat`:

```
main :: IO ()
main = do
  x <- getContents
  putStr x
```

- Die Standardeingabe wird in die Stringvariable `x` gelesen. Danach auf der Standardeingabe ausgegeben.
- Der Typ von **`getContents`** ist **`IO String`**.
- D.h. es wird eine Zeichenkette zurückgegeben, und möglicherweise Ein-/Ausgabe getätigt.
- In Haskell sind Seiteneffekte einer Funktion immer im Typ erkennbar!

Sequenzoperatoren

- Die **do**-Notation ist nur *syntactic sugar* für Sequenzoperatoren.

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

- Damit schreibt sich das Hallo-Welt-Programm so:

```
main = putStrLn "Hello, world!" >> putStrLn "Bye-bye!"
```

- Und das cat-Programm so:

```
main = getContents >>= \ x -> putStr x
```

- Oder noch kürzer so:

```
main = getContents >>= putStr
```

Effektvolle Operationen

- Effektvolle Funktionen können ein Ergebnis mit **return** zurückgeben.

```
return :: a -> IO a
```

- Z.B. eine "geschwätzige" Additionsfunktion, die ihr Ergebnis auf der Standardausgabe "verkündet".

```
noisyPlus :: Integer -> Integer -> IO Integer
noisyPlus x y = do
  let z = x + y
  putStrLn (show z)
  return z
```

- Ohne **do**-Notation sieht das so aus:

```
noisyPlus x y =
  let z = x + y
  in  putStrLn (show z) >>
      return z
```

Ausblick: Monaden

- *Interaktion mit der Welt* (**IO**) ist die allgemeinste Form von Effekt.
- **Interne Effekte** wie Ausnahmen oder veränderlicher Zustand kann man *rein funktional* behandeln.
- Z.B. Partialität mit **Maybe**.

```
division :: Integer -> Integer -> Maybe Integer
division x 0 = Nothing
division x y = Just (x `div` y)
```

- Sequentialisierung ist überladen für **Maybe**, ein **Nothing** (Ausnahme) wird propagiert.

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

- Damit ist auch eine entsprechende **do**-Notation definiert.

Ausblick: Monaden

- Rechnen mit partiellen Ergebnissen:

```
div23 :: Integer -> Maybe Integer
div23 x = do
  y <- division x 2
  z <- division (y - 1) 3
  return (z + 1)
```

- Oder:

```
div23' :: Integer -> Maybe Integer
div23' x =
  division x 2 >>=
  \ y -> division (y - 1) 3 >>=
  \ z -> return (z + 1)
```

Ausblick: Monaden

- Implementierung der Sequenzoperatoren:

```
return :: a -> Maybe a
return a = Just a

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

- Die den Sequenzoperatoren zugehörige Typklasse heißt **Monad**.

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

  ma >> mb = ma >>= \ _ -> mb
```

Zusammenfassung

- Haskell ist eine funktionale Sprache mit **verzögerter Auswertung**.
- Erfreut sich wachsender Beliebtheit für sicherheitskritische Software, z.B. in der Finanzbranche.
- **Effekte** sind durch **Monaden** kontrolliert.
- Effektfreiheit ermöglicht Optimierung durch Compiler und Parallelisierung.
- Typsystem und Parallelisierung Gegenstand aktiver Forschung.