



Funktionale Programming

Woche 11 – Verzögerte Auswertung, Unendliche Listen

René Thiemann

Philipp Dablander Joshua Ocker Michael Schaper Lilly Schönherr Adam Pescoller

Letzte Vorlesung

- `IO a` ist Typ von I/O-Aktionen mit Resultat vom Typ `a`
- `do`-Blöcke führen Sequenzen von I/O-Aktionen aus
- klare Trennung zwischen rein funktionalem und I/O-Code:
 - funktionaler Code kann in I/O eingebettet werden: `return :: a -> IO a`
 - die andere Richtung ist nicht möglich: es gibt keine Funktion vom Typ `IO a -> a`
- `ghc` compiliert Programm, in dem eine `main :: IO ()` Funktion in Modul `Main` vorkommt
- Beispiel Anwendung: Vier Gewinnt
 - Benutzeroberfläche: I/O-Code
 - Logik von Vier Gewinnt: rein funktional

Monaden

- Bind (`>>=`), `return`, und `do`-Notation sind **nicht** auf I/O eingeschränkt
- es gibt ein allgemeines Konzept von **Monaden**
- Beispiel: auch der **Maybe**-Typ ist eine Monade

```
data Expr = Const Double | Div Expr Expr
eval :: Expr -> Maybe Double
eval (Const c) = return c
eval (Div expr1 expr2) = do
  x1 <- eval expr1
  x2 <- eval expr2
  if x2 == 0
    then Nothing
    else return (x1 / x2)
```

- Monaden werden in diesem Kurs nicht weiter behandelt, aber es ist der Grund für die Verwendung des Begriffs **I/O-Monade** in der Haskell Literatur

Auswertungs-Strategien

Reine Funktionen

- eine Funktion ist **rein** (englisch: *pure*), wenn man bei gleicher Eingabe immer das gleiche Resultat erhält, und die Ausführung keine Seiteneffekte hat
- Beispiele von reinen Funktionen
 - Addition
 - Sortierung einer Liste (Rückgabe einer neuen sortierten Liste)
 - ...
- nicht reine Funktionen
 - Sortierung einer Liste (ohne dabei eine neue Liste zu erzeugen)
 - Bestimmung der aktuellen Zeit
 - Bestimmung eines Würfel-Wurfs
 - ...
- rein-funktionale Sprachen erlauben es, nur reine Funktionen zu definieren
- Haskell ist **rein-funktional**

Reine Funktionen und I/O

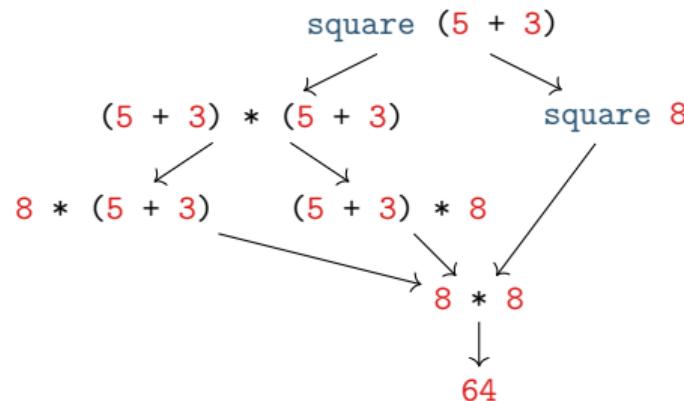
- selbst I/O ist in Haskell rein
- betrachte `main = getLine >>= putStrLn . ("Hello " ++)`
- man könnte meinen, dass das Resultat von der Eingabe abhängt, also nicht rein ist
- aber `main :: IO ()`, also ist der funktionale Wert von `main` eine (kombinierte) Aktion; und diese Aktion ist in der Tat immer gleich:

lese erst eine Eingabe i und gebe dann den Text "Hello i " aus

- alternative Argumentation: interpretiere Typ `IO a` als ein Zustands-Transformer auf der Welt, d.h. als eine Funktion `RealWorld -> (RealWorld, a)`
- Anmerkung: im Rest dieser Vorlesung betrachten wir nur reine Funktionen ohne I/O

Auswertungs-Reihenfolge

- es gibt viele Möglichkeiten, einen Ausdruck auszuwerten; betrachte `square x = x * x`



- in rein funktionalen Sprache hat die Auswertungs-Reihenfolge keinen Einfluß auf die berechneten Normalformen
- Normalform: ein Ausdruck, der nicht weiter ausgewertet werden kann; ein Resultat

Theorem

Wann immer man einen Haskell-Ausdruck auf zwei (unterschiedliche) Arten zu einer Normalform auswerten kann, dann sind die erhaltenen Normalformen identisch.

Auswertungs-Strategien

- Programmiersprachen fixieren Auswertungs-Reihenfolge durch Auswertungs-Strategie
- bekannte Strategien mit Ausdrücken als Bäumen oder als gerichtete Graphen (DAGs)
 - **innermost** (strikt): werte zuerst die Argumente einer Funktion aus

$$\text{square } (\text{square } (5+3)) = \text{square } (\text{square } (\text{square } (5+3))) = \text{square } (\text{square } (\text{square } (8))) = \text{square } (\text{square } (8)) = \text{square } (64)$$

square square *

 | | / \ \ \

 + 8 8 8

 / \ / \ \ \

 5 3 5 3 5 3 5 3

- **outermost** (nicht strikt): wende Gleichung so früh wie möglich an

$$\text{square } (\text{square } (5+3)) = \text{square } (\text{square } (\text{square } (5+3))) = \text{square } (\text{square } (\text{square } (\text{square } (5+3)))) = \text{square } (\text{square } (\text{square } (8))) = \text{square } (\text{square } (8)) = \text{square } (64)$$

square * * *

 | / \ / \ \ \ / \ \ \

 + + + 8 8 8

 / \ / \ \ \ / \ \ \ / \ \ \

 5 3 5 3 5 3 5 3 5 3 5 3

- **lazy evaluation** (nicht strikt): wie outermost + Verwendung von DAGs

$$\text{square } (\text{square } (5+3)) = \text{square } (\text{square } (\text{square } (5+3))) = \text{square } (\text{square } (\text{square } ((5+3)))) = \text{square } (\text{square } (8)) = \text{square } (64)$$

square * *

 | () ()

 + + 8

 / \ / \ \ \

 5 3 5 3

Auswertungs-Strategie von Haskell

- Haskell verwendet Lazy Evaluation (**verzögerte Auswertung**) mit Argument-Reihenfolge von links nach rechts
- Sharing im DAG wird verwendet, wann immer eine Variable mehrfach genutzt wird
- Beispiel Gleichung $f\ x = g\ x + g\ (3 + 5) + x$
 - bei der Auswertung von $f\ (1 + 2) = g\ (1 + 2) + g\ (3 + 5) + (1 + 2)$ werden die beiden Vorkommen von $1 + 2$ geshared: sie nutzen die gleiche Variable x
 - bei der Auswertung von $f\ (3 + 5) = g\ (3 + 5) + g\ (3 + 5) + (3 + 5)$ werden die beiden Vorkommen von $g\ (3 + 5)$ nicht geshared: es war ein Zufall, dass x mit $3 + 5$ substituiert wurde und diese Gleichheit wird nicht zur Laufzeit entdeckt
- Compiler kann weiteres Sharing erzeugen; z.B. wird in Funktions-Definition $f\ x = g\ x + h\ (g\ x)$ das $g\ x$ nur einmal erzeugt und berechnet
- Auswertung von Argumenten einer Ausdrucks $f\ \text{expr1} \dots \text{exprN}$ wird durch Pattern Matching angestoßen, um z.B. die nutzbaren Gleichungen der Form $f\ \text{pat1} \dots \text{patN} = \text{expr}$ zu bestimmen, siehe Folie 13 und 15 aus Woche 3
- viele eingebaute arithmetische Funktionen benötigen Auswertung aller Argumente, z.B. liefert $(0 :: \text{Integer}) * \text{undefined}$ einen Fehler und nicht etwa 0

Auswertungs-Strategien und Terminierung

- betrachte folgendes Programm

```
three :: Integer -> Integer
```

```
three x = 3
```

```
inf :: Integer
```

```
inf = 1 + inf
```

- Innermost-Auswertung terminiert nicht, d.h., die Auswertung läuft unendlich

```
three inf = three (1 + inf) = three (1 + (1 + inf)) = ...
```

- Outermost- und Lazy-Strategie sind sofort fertig

```
three inf = 3
```

Theorem

- wenn die Auswertung eines Ausdruck für irgendeine Strategie terminiert, dann terminiert auch die Auswertung mit Outermost- und Lazy-Strategie
- wenn die Auswertung eines Ausdrucks mit Innermost-Strategie terminiert, dann terminiert die Auswertung für jede Strategie

Vergleich der Auswertung-Strategien

- Innermost
 - einfach zu verstehen
 - einfach zu implementieren: Argumente einer Funktionen sind immer Werte
 - teilweise wird zu viel berechnet, wenn Argumente für Resultat nicht benötigt werden
 - wird in vielen funktionalen Sprachen verwendet
- Lazy Evaluation
 - schwieriger zu verstehen
 - Auswertung ist aufwändiger zu implementieren:
man muss **Thunks** übergeben, also Ausdrücke, die noch ausgewertet werden können
 - Verwaltung von Thunks ist Mehraufwand bei der Berechnung
 - ermöglicht es, einfach mit unendlichen Daten umzugehen
 - wird in Haskell verwendet

Endrekursion und Strikte Auswertung

Rekursions-Arten

- **Rekursion**: eine Funktion, die sich selbst aufruft ist
- **verschränkte Rekursion**: Funktionen, die sich gegenseitig aufrufen

```
even n | n == 0      = True
        | otherwise = odd (n - 1)
odd n  | n == 0      = False
        | otherwise = even (n - 1)
```

- **geschachtelte Rekursion**: rekursive Aufrufe finden innerhalb rekursiver Aufrufe statt

```
ack n m | n == 0 = m + 1
        | m == 0 = ack (n - 1) 1
        | otherwise = ack (n - 1) (ack n (m - 1))
```

- **lineare Rekursion**: maximal ein rekursiver Aufruf (pro if-then-else Fall)

- `fib n | n >= 2 = fib (n - 1) + fib (n - 2)` ✗
- `length (x : xs) = 1 + length xs` ✓
- `f x = if even x then f (x `div` 2) else f (3 * x + 1)` ✓

- **Endkursion** und **guarded Recursion** werden im Detail besprochen

Endrekursion (Tail Recursion)

- Endrekursion ist eine besondere Form der linearen Rekursion
- zusätzliche Anforderung
 - rekursive Aufrufe finden nur ganz außen statt,
bzw. ganz außen innerhalb eines then- oder else-Ausdrucks
- Beispiele
 - `length (x : xs) = 1 + length xs` X
 - `f x = if even x then f (x `div` 2) else f (3 * x + 1)` ✓
- Vorteil der Endrekursion
 - man muss keine weiteren Funktionsaufrufe nach dem rekursiven Aufruf durchführen
 - Konsequenz: Endrekursive Funktionen können einfach als Schleife implementiert werden
 - Speicherplatz sparend

Beispiel: Vorteil der Endrekursion

- linear, aber nicht endrekursiv

```
sumRec 0 = 0
```

```
sumRec n = n + sumRec (n - 1)
```

```
sumRec 5 = 5 + sumRec (5 - 1)
```

```
= 5 + sumRec 4 = 5 + (4 + sumRec (4 - 1))
```

```
= 5 + (4 + sumRec 3) = 5 + (4 + (3 + sumRec (3 - 1))) = ...
```

```
= 5 + (4 + (3 + (2 + (1 + 0)))) = ... = 15 -- linear Platzbedarf
```

- endrekursiv; nutzt **Akkumulator** um Zwischenergebnisse zu speichern

```
sumTr n = aux 0 n where
```

```
aux acc 0 = acc
```

```
aux acc n = aux (acc + n) (n - 1)
```

```
sumTr 5
```

```
= aux 0 5 = aux (0 + 5) (5 - 1)
```

```
= aux 5 4 = aux (5 + 4) (4 - 1)
```

```
= aux 9 3 = ... = 15
```

-- konstanter Platzbedarf, Implementierung durch Schleife mit 2 Variablen

Problem bei Endrekursion durch Lazy Evaluation

```
sumTr n = aux 0 n where
  aux acc 0 = acc
  aux acc n = aux (acc + n) (n - 1)
```

- Auswertung von `sumTr` auf voriger Folie nutzt Innermost-Strategie
- bei Lazy Evaluation werden `acc` und `n` nur auf Bedarf ausgewertet
- führt zu linearem Speicherbedarf bei `sumTr`

```
sumTr 5           -- with lazy evaluation
= aux 0 5
= aux (0 + 5) (5 - 1)
= aux (0 + 5) 4
= aux ((0 + 5) + 4) (4 - 1)
= ...
= aux (((((0 + 5) + 4) + 3) + 2) + 1) 0
= (((0 + 5) + 4) + 3) + 2) + 1 = ... = 15
```

Erzwingen der Auswertung

- Haskell Funktion, um eine Auswertung zu erzwingen: `seq :: a -> b -> b`
- Auswertung von `seq x y` wertet erst `x` zu WHNF aus und gibt dann `y` zurück
- **WHNF**: weak head normal form
- Ausdruck `e` ist in WHNF gdw. er eine der folgenden drei Formen hat
 - `e = C expr1 ... exprN` für einen Konstruktor `C` (Konstruktor Anwendung)
 - `e = f expr1 ... exprN` wenn die definierenden `f`-Gleichungen `M > N` Argumente haben, d.h., sie sind von der Form `f pat1 ... patM = expr` (zu wenige Argumente)
 - `e = \ pat1 ... patN -> expr` (λ -Abstraktion)
- Beispiel
 - in WHNF: `True, 7.1, (5+1) : [1] ++ [2], ((): undefined : undefined, (++), (++ undefined), \ x -> undefined`
 - nicht in WHNF: `[1] ++ [2], (\ x -> x + 1) (1 + 2), undefined ++ undefined`
 - Auswertung: `let x = 1 + 2 in seq x (f x)`
= `seq (1 + 2) (f (1 + 2))` -- with 1 + 2 shared
= `seq 3 (f 3)` -- seq enforced evaluation of argument
= `f 3 = ...` -- evaluation of f 3 continues

Beispiel Anwendung von seq

- löse das Platzproblem bei der Endrekursion, durch das Erzwingen der Auswertung des Akkumulators

```
sumTrSeq n = aux 0 n where
  aux acc 0 = acc
  aux acc n = let accN = acc + n in seq accN (aux accN (n - 1))

  sumTrSeq 5
= aux 0 5
= let accN = 0 + 5 in seq accN (aux accN (5 - 1))                                -- 0 + 5 is shared
= seq (0 + 5) (aux (0 + 5) (5 - 1))                                              -- and evaluated
= seq 5 (aux 5 (5 - 1))
= aux 5 (5 - 1)
= aux 5 4                                -- pattern matching triggers evaluation
= let accN = 5 + 4 in seq accN (aux accN (4 - 1))                                -- 5 + 4 is shared
= seq (5 + 4) (aux (5 + 4) (4 - 1))                                              -- and evaluated
= seq 9 (aux 9 (4 - 1))
= aux 9 (4 - 1)                           -- same structure as above
= ... = 15                                -- constant space
```

Erzwingen der Auswertung ... Fortgesetzt

- neben `seq` gibt es andere Möglichkeiten, die Auswertung zu erzwingen
- **strikte Bibliotheksfunktionen** wie z.B. eine strikte Variante von `foldl`:
`foldl' :: (b -> a -> b) -> b -> [a] -> b`
`length = foldl' (\ x _ -> x + 1) 0`
- Pattern Matching mit **Bang Patterns**, um die Auswertung zu erzwingen, z.B.
`aux acc n = let !accN = acc + n in aux accN (n - 1)`
- **strikte Datentypen**
- weitere Details: <https://downloads.haskell.org/~ghc/latest/docs/html/users-guide/exts/strict.html>

Lazy Evaluation und Unendliche Listen

Guarded Recursion

- jeder rekursive Aufruf befindet sich innerhalb eines Konstruktor-Aufrufs
- alternative Bezeichnung: **Endrekursion modulo Cons**
- in Haskell ist Guarded Recursion wichtiger als Endrekursion
- Guarded Recursion erlaubt es, das Resultat stückweise zu erzeugen und zu konsumieren; im Gegensatz dazu wird bei der Endrekursion das Ergebnis erst zum Schluss offenbart
- Beispiele von Guarded Recursion

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

✓

- `reverse xs = revAux xs [] where`
`revAux [] ys = ys`

✗

- `revAux (x : xs) ys = revAux xs (x : ys)`

- `enumFrom x = x : enumFrom (x + 1)`

✓

- Anmerkungen zu `enumFrom`

- `enumFrom` wurde vereinfacht, die eingebaute `enumFrom` Funktion funktioniert für alle Typen der `Enum`-Klasse, z.B., `Int`, `Char`, `Integer`, `Double`, ... und verhindert Überläufe
- Syntaktischer Zucker: `[x..] = enumFrom x`

Unendliche Listen

- unendliche Listen \sim **Folgen** von Elementen (auch: Datenstrom)
- Programmierung mit unendlichen Listen: Produktion und Verbrauch von Elementen stückweise zu Beginn der Liste (z.B. durch Guarded Recursion)
- Beispiel: $[x..] = x : [x + 1 ..]$ generiert unendliche Liste
- in Kombination mit Lazy Evaluation führt dies nicht immer zur nicht-Terminierung
- Beispiele

```
take 2 [7..]
= take 2 (7 : [8..])
= 7 : take 1 [8..]
= 7 : 8 : take 0 [9..]
= [7, 8]
```

```
takeWhile (< 95) $ map (\ x -> x * x) [0..]
= ... = [0,1,4,9,16,25,36,49,64,81]

filter (< 100) $ map (\ x -> x * x) [0..]
= ... = [0,1,4,9,16,25,36,49,64,81] -- interrupted
```

Lazy Evaluation und Unendliche Datenstrukturen fördern die Modularität

- schreibe kleine Funktionen für spezifische Aufgaben
- nutze dabei potentiell unendliche Datenstrukturen
- Beispiel für Lazy Evaluation: finde Index des ersten Listenelements, dass ein Prädikat erfüllt

- Funktion `firstIndex :: (a -> Bool) -> [a] -> Int`

```
firstIndex p = fst . head . filter (p . snd) . zip [0..]
```

- (lazy) Auswertung (ohne hierbei explizit die Expansion von (.) und (\$) zu zeigen)

```
firstIndex (== 1) [1..9]
= fst . head . filter ((== 1) . snd) $ zip [0..] [1..9]
= fst . head . filter ((== 1) . snd) $ (0,1) : zip [1..] [2..9]
= fst . head $ (0,1) : filter ((== 1) . snd) $ zip [1..] [2..9]
= fst (0,1)
= 0
```

- ohne Lazy Evaluation würde die **vollständige** Liste durchlaufen
(z.B., Berechnung der Länge und Hinzufügen der Indizes)
- Anmerkung: `firstIndex` funktioniert für beliebige Listen: endliche und unendliche

Sieb des Eratosthenes

- Ziel: generiere Liste **aller** Primzahlen
- Algorithmus
 1. beginne mit Liste aller natürlichen Zahlen beginnend mit 2
 2. markiere erste Zahl x der Liste als Primzahl
 3. entferne alle Vielfachen von x
 4. gehe zurück zu Schritt 2

- in Haskell

```
primes :: [Integer]
primes = sieve [2..] where
  sieve (x : xs) = x : sieve (filter (\ y -> y `mod` x /= 0) xs)

> take 1000 primes          -- the first 1000 primes
> takeWhile (< 1000) primes -- all primes below 1000
> primes !! 1000           -- prime number #1001
```

Zusammenfassung

- in rein funktionalen Sprachen wie Haskell hängt das Resultat nicht von der Auswertungs-Strategie ab
- es gibt unterschiedliche Formen von Rekursion
- **Endrekursion** ist oft effizient, da es als Schleife implementiert werden kann
- **seq** kann eine **strikte Auswertung erzwingen**
(in Haskell bei Endrekursion oft für Akkumulatoren eingesetzt)
- **Lazy Evaluation** ermöglicht den Umgang mit unendlichen Listen
- **Guarded Recursion** ist wichtig für Algorithmen mit unendlichen Listen
- **unendliche Listen** erlauben eine natürliche Formulierung einiger Algorithmen
(ohne auf Randbedingungen achten zu müssen)