Praktische Informatik 3: Funktionale Programmierung

Vorlesung 3 / 07. November 2023

Rekursion, Listen, Auswertung

Thomas Barkowsky

Wintersemester 2023/24

Übersicht

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- Testen und Qualitätssicherung
- I/O, Aktionen und Zustände
- Monaden
- Domänenspezifische Sprachen
- Funktionale Programmierung in der Praxis

heute in dieser Vorlesung...

- Rekursion (statt Schleifen)
  - Listen, multiple Rekursion, wechselseitige Rekursion, ...
- *List comprehensions* / Listenumschreibungen
  - *Guards* und Generatoren
- Auswertung von Ausdrücken in Haskell
  - Striktheit und *Lazy evaluation*

Rekursion

Rekursion statt Schleifen

- Definition von Funktionen durch sich selbst
- Beispiel Fakultät:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Basis

Rekursion

  - Basisfall: 1 als neutrales Element der Multiplikation
- Viele Standardfunktionen in Haskell rekursiv definiert

Rekursion mit Listen

- Beispiel `reverse` (*standard prelude*):

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- Einfügen eines Elements in eine sortierte Liste:

```
insert :: a -> [a] -> [a]
insert x []     = [x]
insert x (y:ys) | x<=y      = x : y : ys
                | otherwise = y : insert x ys
```

Rekursion über mehrere Argumente

- Beispiel `zip` (*standard prelude*):

```
Prelude> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _          = []
zip _ []          = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- Zwei Basisbedingungen, da jede der Listen leer sein kann

Multiple Rekursion

- Funktion mehrfach in ihrer eigenen Definition verwendet
- Beispiel: Berechnung der n-ten Fibonacci-Zahl
  - 0, 1, 1, 2, 3, 5, 8, …

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

Wechselseitige Rekursion

Zwei oder mehr Funktionen rufen sich gegenseitig auf
Beispiel: `odd` und `even`

```
even :: Int -> Bool
even 0 = True
even n = odd (n-1)
```

```
odd :: Int -> Bool
odd 0 = False
odd n = even (n-1)
```

```
even 3 → odd 2 → even 1 → odd 0 → False
even 2 → odd 1 → even 0 → True
```

In *standard prelude* aus Effizienzgründen **nicht** so realisiert!

sondern so:

```
even n = n `rem` 2 == 0
```

Unterschied `rem` vs. `mod`?

`rem` vs. `mod`

- Abhängig von der Definition von `quot` und `div`:

  ‘quot‘ is integer division truncated toward zero, while the result of ‘div‘ is truncated toward negative infinity. *(Haskell 2010 Language Report 6.4.2)*

- Definition von `rem` und `mod`:

```
(x `quot` y)*y + (x `rem` y) == x
(x `div`  y)*y + (x `mod` y) == x
```

(ebd.)

Rekursion:

- “Defining recursive functions is like riding a bicycle:
  - it looks easy if somene else is doing it,
  - may seem impossible when you first try to do it yourself,
  - but becomes simple and natural with practice.”

Graham Hutton

- Definition rekursiver Funktionen in fünf Schritten:
  - 1. Typdefinition
  - 2. Auflistung der relevanten Fälle
  - 3. Definition der Basisfälle
  - 4. Definition der übrigen Fälle
  - 5. Verallgemeinern und vereinfachen

Schritt 1: Typdefinition

- Zwei Beispiele: `drop` und `init`

```
Prelude> drop 2 "impossible"
"possible"
```

```
Prelude> init [1,2,3,4]
[1,2,3]
```

- Typdefinition:

```
drop :: Int -> [a] -> [a]
```

```
init :: [a] -> [a]
```

Schritt 2: Auflistung der relevanten Fälle

- `drop`: vier Fälle, da zwei Argumente
- `init`: nur ein Fall, da `init` für leere Liste nicht definiert

```
drop :: Int -> [a] -> [a]
```

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

```
init :: [a] -> [a]
```

```
init (x:xs) =
```

Schritt 3: Definition der Basisfälle

- `drop`: `drop n [] = []` definiert zur Fehlervermeidung
- `init`: Fall für xs = leere Liste (`null xs`) berücksichtigen
  (d.h. Liste enthält nur (noch) ein Element)

```
drop :: Int -> [a] -> [a]
```

```
drop 0 []     = []
drop 0 (x:xs) = (x:xs)
drop n []     = []
drop n (x:xs) =
```

```
init :: [a] -> [a]
```

```
init (x:xs) | null xs   = []
            | otherwise =
```

Schritt 4: Definition der übrigen Fälle

```
drop :: Int -> [a] -> [a]
```

```
drop 0 []     = []
drop 0 (x:xs) = (x:xs)
drop n []     = []
drop n (x:xs) = drop (n-1) xs
```

```
init :: [a] -> [a]
```

```
init (x:xs) | null xs   = []
            | otherwise = x : init xs
```

Schritt 5: Verallgemeinern und vereinfachen

- `drop 0`: zwei Fälle zusammenfassen
- `drop n`: nicht verwendete Variablen durch *wildcards* ersetzen

```
drop :: Int -> [a] -> [a]
```

```
drop 0 []     = []
drop 0 (x:xs) = (x:xs)
drop n []     = []
drop n (x:xs) = drop (n-1) xs
```

```
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = drop (n-1) xs
```

Schritt 5: Verallgemeinern und vereinfachen

- `init`: Definition mit *pattern matching* und *wildcard*

```
init :: [a] -> [a]
```

```
init (x:xs) | null xs   = []
            | otherwise = x : init xs
```

```
init [_]    = []
init (x:xs) = x : init xs
```

List Comprehensions

List Comprehensions?

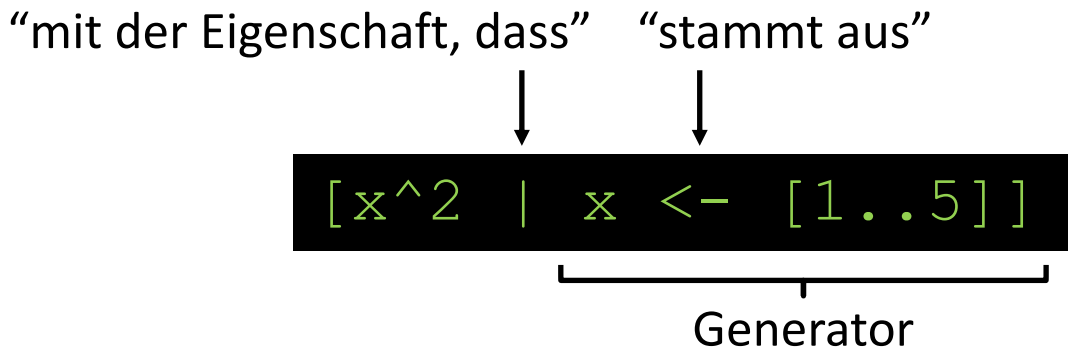
- *comprehension*: the act or process of *comprising* *(merriam-webster.com)*
- *to comprise* = to compose, to constitute *(merriam-webster.com)*
- "Listenumschreibungen" (???)

- Ok, worum geht es?
- Beispiel: M = $\{ x^2 \mid x \in \{1..5\} \} = \{1, 4, 9, 16, 25\}$

- in Haskell:

```
Prelude> [x^2 | x <- [1..5]]
[1,4,9,16,25]
```

List Comprehensions: Beispiele



- ... nicht nur **ein** Generator:

```
Prelude> [(x,y) | x<-[1,2,3], y<-[4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

- ... Reihenfolge der Generatoren:

```
Prelude> [(x,y) | y<-[4,5], x<-[1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Weitere Beispiele

- Voneinander abhängige Generatoren (Liste geordneter Paare):

```
Prelude> [(x,y) | x<-[1..3], y<-[x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

- Mögliche Definition von `concat`:

```
concat :: [[a]] -> [a]
concat xss = [x | xs<-xss, x<-xs]
```

- oder `lenght`:

```
length :: [a] -> Int
length xs = sum [1 | _<-xs]
```

Guards als Filter für Generatoren

- Beispiel:

```
> [x | x<-[1..20], even x]
[2,4,6,8,10,12,14,16,18,20]
```

- oder auch:

```
teiler :: Int -> [Int]
teiler n = [x | x<-[1..n], n `mod` x == 0]
```

```
> teiler 36
[1,2,3,4,6,9,12,18,36]
> teiler 13
[1,13]
```

Auswertung: strikt vs. *lazy*

noch einmal: Auswertung von Ausdrücken

- Beispiel:

```
isOdd n = mod n 2 == 1
```

```
isOdd (1 + 2)
```

- **strikte** Auswertung:

```
isOdd (1 + 2)
→  isOdd 3
→  mod 3 2 == 1
→  1 == 1
→  True
```

  - Argumente werden **vor** Anwendung der Funktion ausgewertet
  - *call-by-value*
  - z.B. Java, C, ...
- Das läuft in Haskell anders!

Nicht-strikte Auswertung in Haskell

- in `isOdd (1 + 2)` wird `(1 + 2)` zunächst **nicht** zu `3` ausgewertet
- stattdessen:
  Auswertung wird zurückgestellt, bis Wert tatsächlich benötigt wird
- "*lazy evaluation*"
- Speicherung als ***thunk*** (noch auszuwertender Ausdruck)
- **Falls Wert nicht benötigt wird, findet auch keine Auswertung statt**
- Wofür ist das gut?
- Exkurs: Konfluenz, Terminierung, Normalform

Ein weiteres Beispiel

```
inc :: Int -> Int
inc n = n + 1
```

```
inc (2 * 3)
```

```
→  inc 6
→  6 + 1
→  7
```

```
→  (2 * 3) + 1
→  6 + 1
→  7
```

Auswertung
von **innen** nach **außen**
innermost *evaluation*

Auswertung
von **außen** nach **innen**
outermost *evaluation*

Innermost vs. *outermost evaluation*

- *Innermost evaluation*:
  - Argumente **vor** Funktionsanwendung vollständig ausgewertet
  - *call by value* (**strikte** Auswertung)
- *Outermost evaluation*:
  - **Funktionsanwendung vor Auswertung der Argumente** möglich
  - *call by name* (auch: *call by need*, wg. verzögerter Auswertung)
- In Haskell führt jede Auswertungsreihenfolge eines Ausdrucks zum selben Ergebnis, **wenn diese terminiert**
  - **Konfluenz:** Der Zeitpunkt der Anwendung einer Funktion auf ihre Parameter hat keinen Einfluss auf das Ergebnis der Berechnung

Termination und Normalform

- Jede Auswertungsreihenfolge in **terminierenden** funktionalen Programmen führt zum selben Ergebnis: **Normalform**
- **Termination**: es gibt keine unendlichen Auswertungsketten
- **Normalform**: nicht weiter auswertbarer Ausdruck

- Auswertungsstrategie in Haskell ist **nicht-strikt**, sondern ***lazy***
- allerdings...

Striktheit

- Viele Funktionen benötigen vollständig ausgewertete Argumente um ausgewertet werden zu können
  - trotz *outermost evaluation*
  - z.B. Operatoren +, *, ...
- Definition Striktheit:
  eine Funktion f ist **strikt**
  gdw.
  Ergebnis der Anwendung von f ist undefiniert, solange ein Argument undefiniert ist

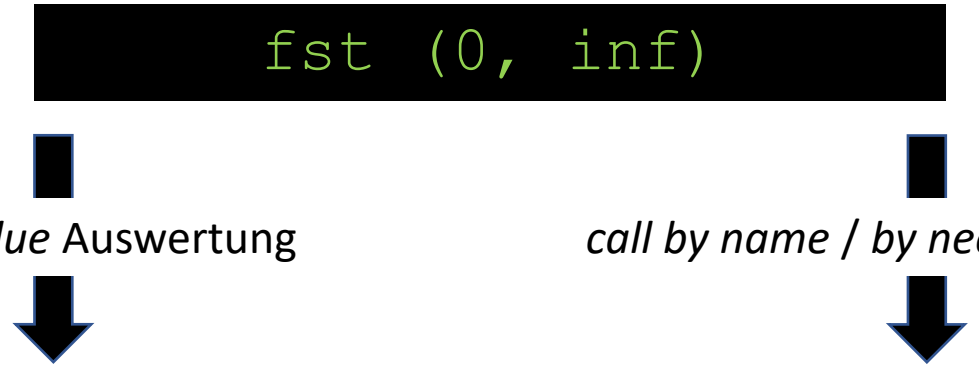
Vorteile nicht-strikter Auswertung?

```
inf :: Int
inf = 1 + inf
```

```
inf
→  1 + inf
→  1 + (1 + inf)
→  1 + (1 + (1 + inf))
→  ...
```

Auswertung und Termination

- Ausdruck:




```
→  fst (0, 1 + inf)
→  fst (0, 1 + (1 + inf))
→  fst (0, 1 + (1 + (1 + inf)))
→  ...
```

```
→  0
```

Auswertung und Termination

- Wichtige Eigenschaft:

> Wenn es eine terminierende Auswertungsreihenfolge für einen gegebenen Ausdruck gibt, dann terminiert *call by name* Auswertung und liefert denselben Wert

- ... wichtig für unendliche Datenstrukturen:

Termininierung in unendlichen Strukturen

- Wie lautet das erste Element einer unendlichen Liste von Einsen?
- Definition:

```
ones :: [Int]
ones = 1 : ones
```

```
*Main> ones
[1,1,1,1,1,1,1,1,1,1,...
```

(unendliche Liste)

```
*Main> head ones
1
```

```
→  head (1 : ones)
→  1
```

Zusammenfassung

- Rekursion als zentrales Prinzip funktionaler Programmierung
  - insbesondere: Listen!
  - Rezept: fünf Schritte
- *List comprehensions*
  - Beschreibung durch Generatoren
  - Filterung durch *Guards*
- Auswertung von Ausdrücken in Haskell
  - nicht-strikte Auswertung = *Lazy evaluation*
  - Termination, Normalform, Striktheit
  - unendliche Datenstrukturen!

nächstes Mal...

- Funktionen höherer Ordnung
  - Funktionen als Argumente von Funktionen
  - *Currying*
  - Funktionen für Listen: `map`, `filter`, etc.
  - Faltung von Listen: `foldr` und `foldl`
  - Lambda-Ausdrücke
  - Funktionskomposition