

Unterabschnitt 3.1.8

λ Abstraktion

Der Bezeichner f ist selbst ein *Wert*:

```
f x = 3 * x + 1
g = f
g 3
```

Wir haben eine Notation für diesen Wert:

```
g = \ x -> 3 * x + 1
```

Das grundlegende Muster für die *Lambda-Abstraktion* ist:

```
\ variable -> expression
```

Wir nennen dies eine *anonyme Funktion*.

Funktionsabstraktion

Einfachen Syntax haben wir schon benutzt:
Mathematik:

$$f(x) = 3x + 1 \quad (1)$$

Haskell:

```
f(x) = 3 * x + 1
```

Haskell (Klammern sind optional):

```
f x = 3 * x + 1
```

λ Abstraktion (2)

Wie für alle Werte funktioniert die Evaluierung durch Ersetzen:

```
f = \ x -> 3 * x + 1
f 7
(\ x -> 3 * x + 1) 7
```

λ Abstraktion (3)

Evaluierung von Funktionen:

- evaluiere das Argument
- **bind**e das Argument an die Variable
- evaluiere den Funktionskörper und ersetze die Variable im Funktionskörper durch den gebundenen Wert

©2010

TU Kaiserslautern

245

Currying

Wie bekommen wir Funktionen mehrerer Variablen?

```
sum = \ x -> ( \ y -> x + y )
sum 3 4
```

- die äußere Funktion hat als Ergebnis eine Funktion, die x ihrem Argument zuzählt
- die innere Funktion zählt x ihrem Argument zu und liefert das Ergebnis

Dieser Ansatz für Funktionen mehrerer Variablen heißt **Currying**.

©2010

TU Kaiserslautern

247

Funktionen als Werte

Funktionen als Werte treten oft natürlich in der Numerik auf:

```
integral lo hi step f =
  if lo>=hi then 0.0
  else (f lo)*step + integral (lo+step) hi step f
```

Die `integral` Funktion ist eine Funktion, die eine numerische Funktion einer Variablen nimmt und integriert:

```
g x = x**2
result = integral (-1.0) 1.0 1e-5 g
```

©2010

TU Kaiserslautern

246

Currying (2)

Haskell hat Kurzformen für Currying:

```
f = \ x -> \ y -> x + y
f = \ x y -> x + y
f x y = x + y
```

©2010

TU Kaiserslautern

248

Beispiel: Lambda Abstraktion für boolesche Werte

Lambda-Abstraktionen sind vollwertige Datenstrukturen.
Definieren wir einen Datentyp für boolesche Werte:

```
true = \ x y -> x
false = \ x y -> y
```

Jetzt können wir `if...then..else` definieren:

```
ite b x y = b x y
```

Und damit dann alle anderen Funktionen:

```
equals x y = ite x (ite y true false) (ite false true)
and x y = ite x (ite y true false) false
or x y = ite x true (ite y true false)
```

Guards und Muster (2)

Fallunterscheidungen und Pattern Guards sind einfach nur
Abkürzungen für `if...then...else`

```
f x
| x == 0 = 0
| x == 1 = 1
| otherwise = f (x-1) + f (x-2)
```

```
f = \ x ->
  if x==0 then 0
  else if x==1 then 1
       else f (x-1) + f (x-2)
```

Guards und Muster

Haskell hat noch weitere Abkürzungen:
Fallunterscheidung und Muster:

```
f 0 = 0
f 1 = 1
f x = f (x-1) + f (x-2)
```

Pattern Guards (Wächter)

```
f x
| x == 0 = 0
| x == 1 = 1
| otherwise = f (x-1) + f (x-2)
```

Typendeklarationen

Haskell hat Typinferenz:

```
> let f x = x=="hello"
> :type f
f :: [Char] -> Bool
>
```

Wir können aber auch explizit deklarieren:

```
f :: [Char] -> Bool
f x = x=="hello"
```

Typendeklarationen (2)

Warum explizit deklarieren?

- Design / Entwicklung
- Dokumentation
- Vermeidung von Überladungsfehlern
- bessere Fehlermeldungen

©2010

TU Kaiserslautern

253

Typendeklarationen (3)

In Haskell können Typendeklarationen das Verhalten eines Programmes ändern (Überladungsfehler):

```
f x = 2^63 * x
g :: Int -> Int
g x = 2^63 * x

> f 10
92233720368547758080
> g 10
0
```

©2010

TU Kaiserslautern

254

Typendeklarationen (4)

```
poly a b c x = a * x**2 + b * x + c
location x = poly 3.0 4.0 x
source x = location x - 3.0
main = print (source 5.0)
```

```
/home/mb/test.hs:4:14:
No instance for (Fractional (t -> t))
arising from a use of `source' at /home/mb/test.hs:4:14-23
Possible fix: add an instance declaration for (Fractional (t -> t))
In the first argument of `print', namely `source 5.0'.
In the expression: print (source 5.0)
In the definition of `main': main = print (source 5.0)
```

©2010

TU Kaiserslautern

255

Typendeklarationen (5)

```
poly :: Float -> Float -> Float -> Float -> Float -> Float
poly a b c x = a * x**2 + b * x + c

location :: Float -> Float
location x = poly 3.0 4.0 x

source :: Float -> Float
source x = location x - 3.0

main = print (source 5.0)
```

Jetzt wird der Fehler an der richtigen Stelle diagnostiziert:

```
/home/mb/test.hs:5:13:
Couldn't match expected type `Float'
against inferred type `Float -> Float'
In the expression: poly 3.0 4.0 x
In the definition of `location': location x = poly 3.0 4.0 x
```

©2010

TU Kaiserslautern

256

Arten von Rekursion

Rekursion ist eine Funktionsdefinition, die sich auf sich selbst bezieht.

- schon mehrfach verwendet
- funktioniert wie in der Mathematik
- Evaluierung durch Einsetzen

Wir unterscheiden verschiedene häufige Arten von Rekursion:

©2010

TU Kaiserslautern

257

Arten von Rekursion (3)

linear rekursiv: nur ein Aufruf in jeder Fallunterscheidung

```
factorial 1 = 1
factorial n = n * factorial (n-1)
```

endrekursiv / tail recursive: nur ein Aufruf, und der ist außen in der Fallunterscheidung

```
facrep result 1 = result
facrep result n = facrep (result*n) (n-1)
factorial n = facrep 1 n
```

Warum?

- endrekursive Funktionen sind effizienter
- Endrekursion entspricht Schleifen

©2010

TU Kaiserslautern

259

Arten von Rekursion (2)

direkt rekursiv / direct recursive

```
factorial n = if n==1 then 1 else n * factorial (n-1)
```

verschränkt rekursiv / mutually recursive

```
even n | n < 0 = undefined
even n = (n == 0) || odd (n-1)
odd n = if n == 0 then False else even (n-1)
```

©2010

TU Kaiserslautern

258

Arten von Rekursion (4)

geschachtelt rekursiv : $f(\dots f(\dots))$

```
f a b n = f a (f a (b-1) n) (n-1)
```

kaskadenartig rekursiv : $h(\dots f(\dots) \dots f(\dots) \dots)$

```
fib n = (+) (fib (n-1)) (fib (n-2))
```

Diese Rekursionsformen können sehr ineffizient sein und oft umgeschrieben werden.

©2010

TU Kaiserslautern

260

Arten von Rekursion (5)

Langsam:

```
fib 0 = 0
fib n = fib (n-1) + fib (n-2)
```

Schnell:

```
fibrep n0 n1 0 = n0
fibrep n0 n1 1 = n1
fibrep n0 n1 n = fibrep n1 (n0+n1) (n-1)
fibfast n = fibrep 0 1 n
```

Unterabschnitt 3.1.9

Listen

Computational Universality

- Die Menge der rekursiven Funktionen ist **berechnungsvollständig** (*computationally universal*).
- Rekursive Funktionsdeklarationen können als eine Gleichung mit einer Variablen verstanden werden, wobei die *Variable* von einem Funktionstyp ist: Beispiel:
Gesucht ist die Funktion f , die folgende Gleichung für alle $n \in \text{Nat}$ erfüllt:

$$f \ n = \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$$

Allerdings besteht keine Garantie, dass das System eine Lösung findet.

Die Datenstruktur der Listen

Grundfunktionen

```
> [1,2,3]
[1,2,3]
> null []
True
> null [1,2,3]
False
> head [1,2,3]
1
> tail [1,2,3]
[2,3]
```

Die Datenstruktur der Listen (2)

Äquivalent zu...

```
> head [1,2,3]
1
> tail [1,2,3]
[2,3]
```

gibt es...

```
> last [1,2,3]
3
> init [1,2,3]
[1,2]
```

Diese sind aber wesentlich weniger effizient und funktionieren ganz anders. (Versuche zu implementieren.)

Die Datenstruktur der Listen (4)

Zugriff nach Länge und Position:

```
> length [1,2,3]
3
> [1,2,3] !! 2
3
```

Häufig in anderen Sprachen, ineffizient und umständlich in Haskell.

Die Datenstruktur der Listen (3)

Mehrere initiale Elemente:

```
> take 2 [1,2,3]
[1,2]
> drop 2 [1,2,3]
[3]
```

Die Datenstruktur der Listen (5)

Der (:) Konstruktor:

```
> 1:[2,3]
[1,2,3]
> 1:2:3:[]
[1,2,3]
> head (1:[])
1
> tail (1:[])
[]
```

Die Datenstruktur der Listen (6)

Zusammenfügen?

```
> 1 : 2
???
```

```
> [1,2,3] ++ [4,5,6]
???
```

©2010

TU Kaiserslautern

269

Die Datenstruktur der Listen (8)

Typen von Listen

```
Prelude> let x = [1,2]
Prelude> :type x
x :: [Integer]
Prelude> let x = ['a', 'b']
Prelude> :type x
x :: [Char]
Prelude> let x = [3.4,4.1]
Prelude> :type x
x :: [Double]
Prelude>
```

- es gibt Listen von vielen verschiedenen Objekten
- der Listentyp ist **parametrisiert**
- der Type einer Liste von Objekten von Typ `a` wird als `[a]` notiert

©2010

TU Kaiserslautern

271

Die Datenstruktur der Listen (7)

Zusammenfügen von Elementen und Listen

```
> 1 : [2]
[1,2]
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

©2010

TU Kaiserslautern

270

Die Datenstruktur der Listen (9)

Listen erlauben nur einen Type

```
> let l = [3, 'a', 3.1]
???
```

```
> let l = [3, [3]]
???
```

©2010

TU Kaiserslautern

272

Die Datenstruktur der Listen (10)

Haskell Definition

Typ: $[a]$, a ist Typparameter

Funktionen:

```
(==), (/=) :: [a] -> [a] -> Bool  wenn (==) auf a definiert
(:)       :: a -> [a] -> [a]
(+++)     :: [a] -> [a] -> [a]
head, last :: [a] -> a
tail, init  :: [a] -> [a]
null       :: [a] -> Bool
length     :: [a] -> Int
(!!)      :: [a] -> Int -> a
take, drop :: Int -> [a] -> [a]
```

Konstanten:

```
[] :: [a]
```

©2010

TU Kaiserslautern

273

TU Kaiserslautern

275

Die Datenstruktur der Listen (12)

Haskell stellt standardmäßig eine Datenstruktur für Listen bereit, die bzgl. des Elementtyps parametrisiert ist. Typparameter werden üblicherweise geschrieben als a, b, \dots
Dem Typ $[a]$ ist als Wertemenge die Menge aller Listen über Elementen vom Typ a zugeordnet.

©2010

TU Kaiserslautern

275

Die Datenstruktur der Listen (11)

Mathematische Definitionen

Eine **Liste über einem Typ** T ist eine total geordnete Multimenge mit Elementen aus T (bzw. eine Folge, d.h. eine Abb. $\text{Nat} \rightarrow T$).

Eine Liste heißt **endlich**, wenn sie nur endlich viele Elemente enthält.

©2010

TU Kaiserslautern

274

Funktionen über Listen

Summiere alle Zahlen in einer Liste von Zahlen:

```
sum :: [Int] -> Int
sum l = if null l then 0
      else (head l) + sum (tail l)
```

```
> sum [1,2,3]
```

6

©2010

TU Kaiserslautern

276

Funktionen über Listen (2)

Wir können dies mit Mustern schreiben:

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

- diese Muster können Variablen und Konstanten enthalten ist einfach eine Konstante
- wir können sowohl (x:xs), wie auch [a,b,c] verwenden

©2010

TU Kaiserslautern

277

©2010

TU Kaiserslautern

279

Funktionen über Listen (4)

```
[1,2,3] = 1:2:3:[]
        = 1:(2:(3:[]))
        = ((:) 1 ((:) 2 ((:) 3 [])))
```

```
sum (1:2:3:[]) = 1 + sum (2:3:[])
               = 1 + 2 + sum (3:[])
               = 1 + 2 + 3 + sum []
               = 1 + 2 + 3 + 0
               = ((+) 1 ((+) 2 ((+) 3 0)))
```

- die Struktur der Listen als 1:2:3:[] ist rekursiv.
- die entsprechenden Berechnungen sind auch rekursiv
- dies ist wegen der Infixnotation etwas schwer zu erkennen

©2010

TU Kaiserslautern

278

©2010

TU Kaiserslautern

280

Funktionen über Listen (3)

- [...] und (:) sind spezielle Funktionen, die wir **Konstruktoren** nennen
- Konstruktoren erzeugen zusammengesetzte Datenstrukturen konstruieren
- Muster werden aus Konstruktoren, Konstanten und Variablen zusammengesetzt
- wir werden später sehen, wie man neue Konstruktoren definiert

Diese Konstruktoren sind auch Operatoren, aber das brauchen sie nicht zu sein:

```
sum [] = 0
sum ((:) x xs) = x + sum xs
```

©2010

TU Kaiserslautern

278

©2010

TU Kaiserslautern

280

Funktionen über Listen (5)

Überprüfen, ob eine Liste sortiert ist:

```
sorted :: [Int] -> Bool
sorted l = if length l < 2 then True
           else if head l > head (tail l) then False
              else sorted (tail l)
```

```
sorted [] = True
sorted [x] = True
sorted (x:y:l) | x>y = False
sorted (x:xs) = sorted xs
```

```
> sorted [1,2,3]
True
> sorted [1,3,2]
False
```

©2010

TU Kaiserslautern

278

©2010

TU Kaiserslautern

280

Funktionen über Listen (6)

Anhängen von zwei Listen:

```
append :: [a] -> [a] -> [a]
append l1 l2 = if l1 == [] then l2
              else (head l1):(append (tail l1) l2)

append :: [a] -> [a] -> [a]
append [] l = l
append (x:xs) l = x : append xs l

> append [1,2] [3,4]
[1,2,3,4]
```

©2010

TU Kaiserslautern

281

283

TU Kaiserslautern

Funktionen über Listen (7)

Die Elemente einer Liste in umgekehrter Reihenfolge anordnen:

```
rev :: [a] -> [a]
rev l = if null l then []
       else append (rev (tail l)) [head l]

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

> rev [1,2,3]
[3,2,1]
```

©2010

TU Kaiserslautern

282

284

TU Kaiserslautern

Funktionen über Listen (8)

Ineffizient:

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

Besser:

revrep [] l = l
revrep (x:xs) l = revrep xs (x:l)
rev l = revrep l []
```

©2010

TU Kaiserslautern

283

283

TU Kaiserslautern

Funktionen über Listen (9)

Zusammenhängen der Elemente einer Liste von Listen:

```
concat :: [[a]] -> [a]
concat xl = if null xl then []
           else append (head xl) (concat (tail xl))

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

> concat [[1,2], [], [3], [4,5,6]]
[1,2,3,4,5,6]
```

©2010

TU Kaiserslautern

284

284

TU Kaiserslautern

Funktionen über Listen (10)

Wende eine Liste von Funktionen vom Typ `Int -> Int` nacheinander auf eine ganze Zahl an:

```
seqappl :: [(Int -> Int)] -> Int -> Int
seqappl l i = if null l then i
             else seqappl (tail l) ((head l) i)
```

```
seqappl [] x = x
seqappl (f:fs) x = seqappl fs (f x)
```

```
> seqappl [(^3), (+1), (*2)] 2
18
```

©2010

TU Kaiserslautern

285

TU Kaiserslautern

287

Unterabschnitt 3.1.10

Tupel

Funktionen über Listen (11)

Bedeutung:

- Informatiker sollten alle diese Beispiele beherrschen
- Wichtig zum Verständnis von Rekursion und rekursiven Datentypen
- Sie sollten sowohl die Version mit Mustern, als auch die einfachen Versionen verstehen
- Später sollten sie ganz äquivalente Funktionen in prozeduralen und objekt-orientierten Sprachen beherrschen
- Sie werden solche Funktionen allerdings selten schreiben; normalerweise verwendet man höhere Abstraktionen (s.u.)

©2010

TU Kaiserslautern

286

Paare

```
> [3, 'a']
???
> (3, 'a')
(3, 'a')
> fst (3, 'a')
3
> snd (3, 'a')
'a'
> let x = (3, 'a')
> :type x
x :: (Integer, Char)
```

©2010

TU Kaiserslautern

288

Paare (2)

```
> (3, 'a', 9.7)
(3, 'a', 9.7)
> fst (3, 'a', 9.7)
???
> let fst3 (a,b,c) = a
> fst3 (3, 'a', 9.7)
3
>
```

- Listen = gleicher Typ, unterschiedliche Länge
- Tupel = verschiedene Typen, gleiche Länge
- Paare = Tupel mit zwei Elementen, fst/snd
- allgemeine Tupel: benutze Muster zum Zugriff

©2010

TU Kaiserslautern

289

Paare (3)

Typ: (a, b) , a, b sind Typparameter

Funktionen:

```
(==), (/=) :: (a, b) -> (a, b) -> Bool
wenn (==) auf a und b definiert
(→, -) :: a -> b -> (a, b)
fst :: (a, b) -> a
snd :: (a, b) -> b
Konstanten: keine
```

Dem Typ (a, b) ist die Menge der geordneten Paare mit Elementen vom Typ a und b zugeordnet.

©2010

TU Kaiserslautern

290

Paare (4)

Mathematische Sicht:
Paare oder **2-Tupel** sind die Elemente des kartesischen Produktes zweier ggf. verschiedener Mengen oder Typen. Der Typ der Paare ist also ein *Produkttyp*.

©2010

TU Kaiserslautern

291

Die Datenstruktur der Einheit

Haskell unterstützt eine Datenstruktur mit einem definierten Wert:

Typ: $()$

Funktionen:
 $(==), (/=) :: () -> () -> Bool$

Konstanten:
 $() :: ()$

Dem Typbezeichner $()$ ist eine einelementige Wertemenge zugeordnet. Der Wert wird als **Einheit** (engl. *unity*) bezeichnet. Die Einheit wird oft als Ergebnis verwendet, wenn es keine relevanten Ergebniswerte gibt.

©2010

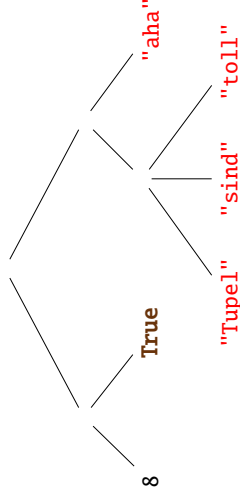
TU Kaiserslautern

292

Beispiel: (Geschachtelte Tupel)

Mit der Tupelbildung lassen sich „baumstrukturierte“ Werte, sogenannte *Tupelsterme*, aufbauen. So entspricht der *Tupelsterm*:
`((8,True), ("Tupel", "sind", "toll"), "aha")`

dem Baum:



Nochmal Currying

Funktionen mehrerer Variablen haben wir mit Currying definiert:

```

sum :: Int -> Int -> Int
sum x y = x + y

sum = \ x -> \ y -> x + y
  
```

Beispiel: (Funktionen auf n -Tupeln)

1. Flache ein Paar von Paaren in ein 4-Tupel aus:

```

ausflachen :: ((a, b), (c, d)) -> (a, b, c, d)
-- nimmt ein Paar von Paaren und liefert 4-Tupel
ausflachen pp = ( fst (fst pp),
                  snd (fst pp),
                  fst (snd pp),
                  snd (snd pp) )
  
```

```
it = ausflachen ( (True,7), ('x',5.6) )
```

Alternative Deklaration mit Mustern:

```
ausflachen ((a, b), (c, d)) = (a, b, c, d)
```

2. Funktion zur Paarbildung:

```

paarung :: a -> b -> (a,b)
paarung lk rk = (lk, rk)
  
```

Nochmal Currying (2)

Mit Tupeln und Mustern können wir auch die mathematische Notation simulieren:

```

sum :: (Int, Int) -> Int
sum (x,y) = x + y

sum = \ p -> (fst p) + (snd p)
  
```

Muster etwas formaler

Muster (engl. *Pattern*) in Haskell sind Ausdrücke gebildet über Bezeichnern, Konstanten und Konstruktoren.

Alle Bezeichner in einem Muster müssen verschieden sein.

Ein Muster M mit Bezeichnern b_1, \dots, b_k **passt auf** einen strukturierten Wert w (engl.: a pattern **matches** a value w), wenn es eine Substitution der Bezeichner b_j in M durch Werte v_j gibt, in Zeichen $M[v_1/b_1, \dots, v_k/b_k]$, so dass

$$M[v_1/b_1, \dots, v_k/b_k] = w$$

Mehr Syntax

Haskell hat noch drei häufige Abkürzungen:

```
> let x = 3 in x*x
9
> x*x where x = 3
9
> case x of 0 -> 0; 1 -> 1; 3 -> 9
9
```

Diese können einfach als Funktionen oder λ -Abstraktionen umgeschrieben werden.

Unterabschnitt 3.1.11

let / where / case

Bezeichnerbindung mit `let`
Die folgenden sind äquivalent:

```
let x = 3 in x*x
(\ x -> x*x) 3
```

Mehr Syntax (3)

Bezeichnerbindung mit `where`
Die folgenden sind äquivalent:

```
x*x where x = 3
```

```
(\ x -> x*x) 3
```

Mehr Syntax (5)

Beobachtungen:

- Die einzelnen Regeln und Syntax sind kompliziert.
- Es ist wichtig diese lesen zu können, aber ihr Gebrauch ist nie notwendig.
- Viele funktionale Sprachen haben diese Konstrukte überhaupt nicht.

Stylistische Empfehlungen:

- Halten Sie Ihre Funktionen sehr kurz.
- Vermeiden Sie diese Konstrukte.
- Wenn Sie eins benötigen, dann am besten `let`.

Mehr Syntax (4)

Fallunterscheidung mit `case`

```
case b of 0 -> 0; 1 -> 1
```

```
helper 0 = 0
```

```
helper 1 = 1
```

```
helper b
```