Praktische Informatik 3: Funktionale Programmierung

Vorlesung 2 / 24. Oktober 2023

Datentypen und Funktionen

Thomas Barkowsky

Wintersemester 2023/24

Universität Bremen

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...

- Datentypen in Haskell
  - Basisdatentypen, Listen, Tupel, Funktionen
  - Polymorphie und Typvariablen
  - Typklassen
- Definition von Funktionen
  - Fallunterscheidung
  - Funktionsdefinition durch *Pattern Matching*
  - Konstruktion von Listen
- Präsentation Bremergy e.V.

Datentypen

Basisdatentypen in Haskell

- `Bool`
  - Werte `True` und `False`

```
Prelude> :t True
True :: Bool
```

- `Char`
  - einzelne Zeichen in Unicode®

```
Prelude> :t 'a'
'a' :: Char
```

- `String`
  - Zeichenketten

```
Prelude> :t "test"
"test" :: [Char]
```

Basisdatentypen in Haskell: Ganze Zahlen

- `Int`
  - Ganze Zahlen als Maschinenworte
  - feste Länge
  - konstanter Speicherbedarf

```
Prelude> 2^32
4294967296
Prelude> 2^63 :: Int
-9223372036854775808
```

type annotation

- `Integer`
  - Ganze Zahlen beliebiger Länge
  - variabler Speicherbedarf
  - flexibler, aber weniger performant

```
Prelude> 2^128
340282366920938463463374607431768211456
```

Basisdatentypen: Fließkommazahlen

- `Float`
  - konstanter Speicherbedarf,
  - einfache Genauigkeit

```
Prelude> sqrt 2 :: Float
1.4142135
```

- `Double`
  - konstanter Speicherbedarf,
  - doppelte Genauigkeit

```
Prelude> sqrt 2
1.4142135623730951
```

Listen

- Elemente desselben Typs
- Anzahl Elemente = Länge
- Leere Liste = 0 Elemente
- Listen können Listen als Elemente enthalten
- Unendliche Listen

```
Prelude> :t ['a','b','c']
['a','b','c'] :: [Char]

Prelude> length ['a','b','c`]
3

Prelude> length []
0

Prelude> :t ["Hund", "Katze", "Maus"]
["Hund", "Katze", "Maus"] :: [[Char]]

Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,27,
28,29,30,31,32,33,34,35,36,37,38,39,
40,41,Interrupted.
```

Nützliche Listen-Funktionen *(standard prelude)*

- `head`: erstes Element einer nicht-leeren Liste
- `tail`: erstes Element aus einer nicht-leeren Liste entfernen
- `!!`: n-tes Element aus einer Liste auswählen
- `take`: erste n Elemente aus einer Liste auswählen
- `drop`: erste n Elemente aus einer Liste entfernen

```
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> [1,2,3] !! 1
2
Prelude> take 2 [1,2,3]
[1,2]
Prelude> drop 2 [1,2,3]
[3]
```

Weitere nützliche Listen-Funktionen

- `length`: Länge einer Liste ausgeben
- `sum`: Summe einer Liste von Zahlen berechnen
- `product`: Produkt einer Liste von Zahlen berechnen
- `++`: zwei Listen verbinden
- `reverse`: eine Liste umkehren

```
Prelude> length [1,2,3]
3
Prelude> sum [1,2,3,4]
10
Prelude> product [1,2,3,4]
24
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
Prelude> reverse [1,2,3]
[3,2,1]
```

Tupel

- **Endliche** Anzahl von Komponenten **unterschiedlichen** Typs
- Leeres Tupel = Stelligkeit (*arity*) 0 -- („*unit type*“)
- Stelligkeit 2: Paar
- Stelligkeit 3: Tripel
- Stelligkeit 1? – nicht zulässig wg. Konflikt mit geklammerten Ausdrücken

```
Prelude> :t (True, 'a', "Spaß")
(True, 'a', "Spaß") :: (Bool, Char, [Char])

Prelude> :t ()
() :: ()
```

Funktionen als Datentyp

```
Prelude> :t not
not :: Bool -> Bool
```

```
Prelude> :t even
even :: Int -> Bool
```

```
Prelude> :t head
head :: [a] -> a
Prelude> :t []
[] :: [a]
Prelude> head []
*** Exception: Prelude.head: empty list
```

a ist Typvariable

partielle Funktion

Polymorphie, Überladung und Typklassen

Polymorphie

- Viele Haskell-Funktionen arbeiten mit verschiedenen Datentypen
- Beispiel: `length`:

```
Prelude> length [1,2,3]
3
Prelude> length [False, False, False]
3
Prelude> length "wort"
4
```

- in der Signatur werden hierfür **Typvariablen** verwendet:

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

Typvariablen

- `length :: [a] -> Int` :
  Für jeden Typ `a` ist die Funktion `length` vom Typ `[a] -> Int`
- Typvariablen (beginnen) immer mit Kleinbuchstaben
- Viele Standardfunktionen in Haskell sind polymorph, z.B.

```
fst  :: (a,b) -> a

head :: [a] -> a

take :: Int -> [a] -> [a]

zip  :: [a] -> [b] -> [(a,b)]
```

Überladene Typen & Typklassen

- *Class constraints* definieren die Anwendbarkeit von Funktionen
- Beispiel:
  Operator "+" funktioniert für
  unterschiedliche Arten von Zahlen:

```
Prelude> 4 + 7
11
Prelude> 1.1 + 3.1
4.2
Prelude> pi + (-2)
1.1415926535897931
```

- Schreibweise: `C a` mit
  - `C`: Name der Typklasse, `a`: Typvariable
- Beispiel: `(+) :: Num a => a -> a -> a`
  - "Für jeden Datentyp aus der Klasse `Num` (numerisch) ist + vom Typ `a -> a -> a`"

Typklassen in Haskell

- Eine Typklasse enthält Datentypen, die mit überladenen Funktionen arbeiten (*Methoden*)
- Die wichtigsten Typklassen (für den Anfang...)
  - `Eq`: Vergleich auf Gleichheit
  - `Ord`: lineare Anordnung
  - `Show`: Darstellung als Strings
  - `Read`: dual zu Show
  - `Num`: numerische Typen
  - `Integral`: numerische Integer-Typen
  - `Fractional`: numerische Nicht-Integer-Typen

Typklasse `Eq`

- Typen, die sich auf Gleichheit testen lassen mit den Methoden:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

- Instanzen: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, sowie hieraus gebildete Listen und Tupel
- Beispiele:

```
Prelude> False == False
True
Prelude> "abc" == "abc"
True
Prelude> [1,2] == [1,2,3]
False
Prelude> ('a', False) == ('a', True)
False
```

Typklasse `Ord`

- Instanzen aus `Eq`, die zusätzlich geordnet sind
- Vergleich durch die Methoden <, <=, >, >=, `min`, `max`
- Instanzen: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, sowie hieraus gebildete Listen und Tupel
- Lexikographische Ordnung bei Strings, Listen, Tupeln
- Beispiele:

```
Prelude> min 'a' 'b'
'a'
Prelude> [1,2,3] < [1,2]
False
Prelude> "elegant" > "elefant"
True
```

Typklasse `Show`

- Typen, die sich in Strings konvertieren lassen mit:

```
show :: a -> String
```

- Instanzen: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, sowie hieraus gebildete Listen und Tupel
- Beispiele:

```
Prelude> show False
"False"
Prelude> show 'a'
"'a'"
Prelude> show 1
"1"
```

Typklasse `Read`

- Dual zu `Show`
- Typen, die sich aus Strings (zurück-)konvertieren lassen mit:

```
read :: String -> a
```

- Instanzen: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, sowie hieraus gebildete Listen und Tupel
- Beispiele:

```
Prelude> read "False" :: Bool
False
Prelude> read "123":: Int
123
```

type annotation

- Explizite Angabe des Zieltyps, sofern nicht aus Kontext inferierbar

Typklasse `Num`

- Alle numerische Typen
- Verarbeitung durch `+`, `-`, `*`, `negate`, `abs`, `signum`
- Beispiele:

```
Prelude> signum (-3)
-1
Prelude> 1 + 2.0
3.0
Prelude> negate (-3)
3
```

- Achtung: negative Zahlen müssen in der Regel geklammert werden
- Division behandelt in Unterklassen `Integral` und `Fractional`

Typklasse `Integral`

- Numerische Integer-Typen (`Int` und `Integer`)
- Integer Division und Rest: `div`, `mod`

```
Prelude> div 7 2
3
Prelude> mod 7 2
1
```

Typklasse `Fractional`

- Numerische Nicht-Integer-Typen (`Float` und `Double`)
- Echte Division und Kehrwert: `/`, `recip`

```
Prelude> 7.0 / 2.0
3.5
Prelude> recip 2.0
0.5
```

Definition von Funktionen

Benennung von Funktionen

- Funktionsbezeichner beginnen mit Kleinbuchstaben
- Danach optional weitere Zeichen:
  - Klein- und Großbuchstaben, Ziffern, Unterstrich, Hochkomma
- Haskell-Schlüsselwörter nicht als Funktionsbezeichner:
  - `case class data default deriving do else foreign if import in infix instance let ...`

Layoutregel für Funktionsdefinitionen

- Definitionen einer Definitionsebene beginnen in gleicher Spalte
  - z.B. in

```
a = b + c
    where
        b = 1
        c = 2
d = a * 2
```

  - sind `b` und `c` lokale Definitionen innerhalb von `a`
- Achtung: Tabulatorzeichen potentiell problematisch
  - GHC wirft Warnung aus
  - besser vermeiden...

Kommentare in Haskell Skripts

- Einzeilige Kommentare

```
-- this line will be ignored by the Haskell compiler
```

- Mehrzeilige Kommentare

```
{-

in this example, five lines will be ignored by GHC

-}
```

Funktionen und Bedingungen

- Konstruktion von neuen Funktionen aus bereits definierten
- Fallunterscheidung in Funktionen:
  - bedingte Ausdrücke

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

  - Verschachtelung möglich

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
               if n==0 then 0 else 1
```

  - `else`-Bedingung immer notwendig!
  - besser als Verschachtelung: *Guarded equations*!

Guarded Equations

- Folge logischer Ausdrücke (*guards*) regelt Auswahl des Resultats

```
abs :: Int -> Int
abs n | n >= 0     =  n
      | otherwise = -n
```

```
signum :: Int -> Int
signum n | n < 0      = -1
         | n == 0     =  0
         | otherwise =  1
```

- bessere Lesbarkeit!
- Bedingungen werden **sequentiell** überprüft
- `otherwise = True` per Definition in *standard prelude*
- `otherwise` ist optional, aber unbedingt empfohlen

Funktionsdefinition mit *Pattern Matching*

- Separate Funktionsrümpfe für unterschiedliche Argumentmuster
- Sequentielle Analyse der *Patterns*
- Beispiel: Definition von `not`:

```
not :: Bool -> Bool
not False = True
not True  = False
```

- Wildcard-Zeichen: "_"
- Beispiel: Konjunktion:
  (aus *standard prelude*)

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
```

Operator

Patterns mit Tupeln

- Verwendung von Tupeln bestimmter Stelligkeit in *Patterns*
- Beispiel: `fst` und `snd` aus *standard prelude*:

```
fst :: (a,b) -> a
fst (x,_) = x
```

```
snd :: (a,b) -> b
snd (_,y) = y
```

- ... und wie ist das bei Listen?

Patterns mit Listen

- geht genauso, z.B.

```
test :: [Char] -> Bool
test ['a',_,_] = True
test _         = False
```

- und für Listen flexibler Länge???
- was sind Listen in Haskell? – Exkurs: der *cons* Operator

Listen: der *cons* Operator (:)

- Listen keine primitiven, sondern komplexe, konstruierte Strukuren
- Listen werden durch sukzessives Anfügen von Elementen erzeugt:
- Beispiel: Liste `[1,2,3]`
  - Start mit leerer Liste:
  - Anfügen eines neuen Elements (`3`):
  - weiteres Element (`2`):
  - und schließlich:
  - vereinfacht (*cons* rechtsassoziativ):

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

- Und jetzt: der *cons* Operator beim Pattern Matching...

Pattern Matching mit *cons*

- neu:

```
test :: [Char] -> Bool
test ('a':_) = True
test _       = False
```

alt:

```
test :: [Char] -> Bool
test ['a',_,_] = True
test _         = False
```

- weiteres Beispiel: `head` und `tail`

```
head :: [a] -> a
head (x:_) = x
```

```
tail :: [a] -> [a]
tail (_:xs) = xs
```

- Achtung: cons Operator geklammert, wg. höchster Priorität für Funktionsanwendung:
  - `head x:_ = x` würde bedeuten: `(head x):_ = x` (???)

Zusammenfassung

- Datentypen in Haskell:
  - Wahrheitswerte, Zeichen(-ketten), Zahlen, Listen, Tupel, Funktionen
- Polymorphie: Typvariablen
- Typklassen und *class constraints*
  - `Eq`, `Ord`, `Show`, `Read`, `Num`, `Integral`, `Fractional`
- Syntax der Funktionsdefinition
  - Benennung, Layout, Kommentare
  - Fallunterscheidung und *Pattern matching*
- Der cons Operator: Listenkonstruktion und *Pattern matching*

nächstes Mal...

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