Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 14.11.2006: Listen, Polymorphie und Rekursion

Christoph Lüth

Inhalt

- Letzte Vorlesung
  - Basisdatentypen
  - Definition von Funktionen durch rekursive Gleichungen
- Diese Vorlesung:
  - strukturierte Typen: Tupel und Listen
  - Pattern matching, Listenkomprehension und primitive Rekursion
- Neue Sprachkonzepte:
  - Polymorphie — Erweiterung des Typkonzeptes

Strukturierte Datentypen: Tupel und Listen

- Strukturierte Typen:
  - Konstruieren aus bestehenden Typen neue Typen.
- Tupel sind das kartesische Produkt:

  `(t1, t2)` = alle Kombinationen von Werten aus `t1` und `t2`.
- Listen sind Sequenzen:

  `[t]` = endliche Folgen von Werten aus `t`

Beispiele

- Modellierung eines Einkaufswagens
  - Inhalt: Menge von Dingen mit Namen und Preis

```
type Item   = (String, Int)
type Basket = [Item]
```

Beispiele

- Modellierung eines Einkaufswagens
  - Inhalt: Menge von Dingen mit Namen und Preis

```
type Item   = (String, Int)
type Basket = [Item]
```

- Punkte, Rechtecke, Polygone

```
type Point   = (Int, Int)
type Line    = (Point, Point)
type Polygon = [Point]
```

Funktionen über Listen und Tupeln

- Funktionsdefinition durch pattern matching:

```
add :: Point-> Point-> Point
add (a, b) (c, d) = (a+ c, b+ d)
```

- Für Listen:
  - entweder leer
  - oder bestehend aus einem Kopf und einem Rest

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

  - Hier hat `x` den Typ `Int`, `xs` den Typ `[Int]`.

Weitere Beispiele

- Gesamtpreis des Einkaufs:

```
total :: Basket -> Int
total [] = 0
total ((name, price):rest) = price + total rest
```

Weitere Beispiele

- Gesamtpreis des Einkaufs:

```
total :: Basket -> Int
total [] = 0
total ((name, price):rest) = price + total rest
```

- Translation eines Polygons:

```
move :: Polygon-> Point-> Polygon
move [] p  = []
move ((x, y):ps) (px, py) =
                (x+ px, y+ py): move ps (px, py)
```

Zeichenketten: `String`

- `String` sind Sequenzen von Zeichenketten:

```
type String = [Char]
```

- Alle vordefinierten Funktionen auf Listen verfügbar.
- Syntaktischer Zucker zur Eingabe:

```
['y','o','h','o'] == "yoho"
```

- Beispiel:

```
count :: Char-> String-> Int
count c []     = 0
count c (x:xs) = if (c== x) then 1+ count c xs
                 else count c xs
```

Beispiel: Palindrome

- Palindrom: vorwärts und rückwärts gelesen gleich
  (z.B. Otto, Reliefpfeiler)

Beispiel: Palindrome

- Palindrom: vorwärts und rückwärts gelesen gleich
  (z.B. Otto, Reliefpfeiler)
- Signatur:

```
palindrom :: String-> Bool
```

- Entwurf:
  - Rekursive Formulierung:
    erster Buchstabe = letzer Buchstabe, und Rest auch Palindrom
  - Termination:
    Leeres Wort und monoliterales Wort sind Palindrome
  - Hilfsfunktionen:
    `last: String-> Char`, `init: String-> String`

Beispiel: Palindrome

- Implementierung:

```
palindrom :: String-> Bool
palindrom []     = True
palindrom [x]    = True
palindrom (x:xs) = (x == last xs)
                   && palindrom (init xs)
```

Beispiel: Palindrome

- Implementierung:

```
palindrom :: String-> Bool
palindrom []     = True
palindrom [x]    = True
palindrom (x:xs) = (x == last xs)
                   && palindrom (init xs)
```

- Kritik:
  - Unterschied zwischen Groß- und kleinschreibung

```
palindrom (x:xs) = (toLower x == toLower (last xs))
                   && palindrom (init xs)
```

  - Nichtbuchstaben sollten nicht berücksichtigt werden.

Listenkomprehension

- Funktionen auf Listen folgen oft Schema:
  - Eingabe generiert Elemente,
  - die getestet und
  - zu einem Ergebns transformiert werden

Listenkomprehension

- Funktionen auf Listen folgen oft Schema:
  - Eingabe generiert Elemente,
  - die getestet und
  - zu einem Ergebns transformiert werden
- Beispiel Palindrom:
  - alle Buchstaben im String `str` zu Kleinbuchstaben.

    ```
    [ toLower c | c <- str ]
    ```
  - Alle Buchstaben aus `str` herausfiltern:

    ```
    [ c | c <- str, isAlpha c ]
    ```
  - Beides zusammen:

    ```
    [ toLower c | c<- str, isAlpha c]
    ```

Listenkomprehension

- Generelle Form:

```
[E | c<- L, test1, ... , testn]
```

- Mit pattern matching:

```
addPair :: [(Int, Int)]-> [Int]
addPair ls = [ x+ y | (x, y) <- ls ]
```

- Auch mehrere Generatoren möglich:

```
[E |c1<- L1, c2<- L2, ..., test1, ..., testn ]
```

Beispiel: Quicksort

- Zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
- sortiere Teilstücke,
- konkateniere Ergebnisse.

Beispiel: Quicksort

- Zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
- sortiere Teilstücke,
- konkateniere Ergebnisse.

```
qsort :: [Int]-> [Int]
qsort [] = []
qsort (x:xs) =
  qsort smaller ++ x:equals ++ qsort larger where
    smaller = [ y | y <- xs, y < x ]
    equals  = [ y | y <- xs, y == x ]
    larger  = [ y | y <- xs, y > x ]
```

Beispiel: Permutation

- Alle Permutationen eines Strings:

```
perms :: String -> [String]
```

- Permutation von `x:xs`
  - `x` an allen Stellen in alle Permutationen von `xs` eingefügt.

```
perms [] = [[]]
perms (x:xs) = [ ps ++ [x] ++ qs
               | rs <- perms xs, (ps, qs) <- splits rs ]
```

- Dabei `splits`: alle möglichen Aufspaltungen

```
splits :: String -> [(String, String)]
splits []     = [("","")]
splits (y:ys) = ([], y:ys) :
                [(y:ps, qs)| (ps, qs) <- splits ys ]
```

Beispiel: Eine Bücherei

Problem: Modellierung einer Bücherei

Datentypen:

- Ausleihende Personen
- Bücher
- Zustand der Bücherei: ausgeliehene Bücher, Ausleiher

```
type Person = String
type Book   = String
type DBase  = [(Person, Book)]
```

Operationen der Bücherei

- Buch ausleihen und zurückgeben:

```
makeLoan :: DBase -> Person -> Book -> DBase
makeLoan dBase pers bk = [(pers,bk)] ++ dBase
```

  - Benutzt `(++)` zur Verkettung von `DBase`

```
returnLoan :: DBase -> Person -> Book -> DBase
returnLoan dBase pers bk
        = [ pair | pair <- dBase ,
                   pair /= (pers,bk) ]
```

- Suchfunktionen: Wer hat welche Bücher ausgeliehen usw. Test.

```
books :: DBase -> Person -> [Book]
books db who = [ book | (pers,book)<- db,
                        pers== who ]
```

Jetzt oder nie — Polymorphie

- Definition von `(++)`:

```
(++) :: DBase->  DBase-> DBase
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs++ ys)
```

- Verketten von Strings:

Jetzt oder nie — Polymorphie

- Definition von `(++)`:

```
(++) :: DBase->  DBase-> DBase
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs++ ys)
```

- Verketten von Strings:

```
(++) :: String-> String-> String
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs++ ys)
```

- Gleiche Definition, aber unterschiedlicher Typ!

  $\rightsquigarrow$ Zwei Instanzen einer allgemeineren Definition.

Polymorphie

- Polymorphie erlaubt Parametrisierung über Typen:

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

  `a` ist hier eine Typvariable.

- Definition wird bei Anwendung instantiiert:

```
[3,5,57] ++ [39, 18]      "hi" ++ "ho"
```

  aber nicht

```
[True, False] ++ [18, 45]
```

- Typvariable: vergleichbar mit Funktionsparameter

Beispiel: Permutation revisited

- Alle Permutationen einer Liste

```
perms' :: [a] -> [[a]]
```

- Permutation von `x:xs`
  - `x` an allen Stellen in alle Permutationen von `xs` eingefügt.

```
perms' [] = [[]]
perms' (x:xs) = [ ps ++ [x] ++ qs
                | rs <- perms' xs, (ps, qs) <- splits' rs ]
```

- Dabei `splits`: alle möglichen Aufspaltungen

```
splits' :: [a] -> [([a], [a])]
splits' []     = [([],[])]
splits' (y:ys) = ([], y:ys) :
                 [(y:ps, qs)| (ps, qs) <- splits' ys ]
```

Polymorphie: Weitere Beispiele

- Länge einer Liste:

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1+ length xs
```

- Verschachtelte Listen “flachklopfen”:

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

- Kopf und Rest einer nicht-leeren Liste:

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

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

  - Undefiniert für leere Liste.

Übersicht: vordefinierte Funktionen auf Listen

`:`	`a-> [a]-> [a]`	Element vorne anfügen
`++`	`[a]-> [a]-> [a]`	Verketten
`!!`	`[a]-> Int-> a`	`n`-tes Element selektieren
`concat`	`[[a]]-> [a]`	“flachklopfen”
`length`	`[a]-> Int`	Länge
`head`, `last`	`[a]-> a`	Erster/letztes Element
`tail`, `init`	`[a]-> [a]`	(Hinterer/vorderer) Rest
`replicate`	`Int-> a-> [a]`	Erzeuge `n` Kopien
`take`	`Int-> [a]-> [a]`	Nimmt ersten `n` Elemente
`drop`	`Int-> [a]-> [a]`	Entfernt erste `n` Elemente
`splitAt`	`Int-> [a]-> ([a], [a])`	Spaltet an `n`-ter Position
`reverse`	`[a]-> [a]`	Dreht Liste um

`zip`	`[a]-> [b]-> [(a, b)]`	Macht aus Paar von Listen Liste von Paaren
`unzip`	`[(a, b)]-> ([a], [b])`	Macht aus Liste von Paaren Paar von Listen
`and`, `or`	`[Bool]-> Bool`	Konjunktion/Disjunktion
`sum`	`[Int]-> Int` (überladen)	Summe
`product`	`[Int]-> Int` (überladen)	Produkt

Siehe Thompson S. 91/92.

Palindrom zum letzten:

```
palindrom' xs = reverse l == l where
                l = [toLower c| c<- xs, isAlpha c]
```

Muster *(pattern)*

Funktionsparameter sind Muster: `head (x:xs) = x`

Ein Muster ist:

- Wert (`0` oder `True`)
- Variable (`x`) - dann paßt alles
  - Jede Variable darf links nur einmal auftreten.
- namenloses Muster (`_`) - dann paßt alles.
  - `_` darf links mehrfach, rechts gar nicht auftreten.
- Tupel `(p1, p2, ... pn)` (`pi` sind wieder Muster)
- leere Liste `[]`
- nicht-leere Liste `ph:pl` (`ph, pl` sind wieder Muster)
- `[p1,p2,...pn]` ist syntaktischer Zucker für `p1:p2:...pn:[]`

Zusammenfassung

- Strukturierte Typen: Listen und Tupel
- Schemata für Funktionen über Listen:
  - Pattern Matching
  - Listenkomprehension
  - primitive Rekursion
- Polymorphie:
  - Abstraktion über Typen durch Typvariablen
- Überblick: vordefinierte Funktionen auf Listen
- Definition von pattern