Vorlesung vom 08.11.2004: Listenkomprehension, Polymorphie und Rekursion

Organisatorisches

Neues Tutorium:

Di	8 – 10	MZH 1380	Wolfgang Machert	36
		GW2 B 1410	Christian Maeder	
	17 – 19	MZH 1380	Diedrich Wolter	23
Mi	8 – 10	MZH 6240	Hennes Märtins	26
		MZH 7250	Shi Hui	25
	13 – 15	MZH 6240	Klaus Lüttich	22
		MZH 1380	Dennis Walter	15
		~~MZH 7230~~	~~Christian Maeder~~	15
Mi	17 – 19	MZH 1380	Kai-Florian Richter	13

(Idealgröße: 22)

Inhalt

- Letzte Vorlesung
  - Basisdatentypen, strukturierte Typen: Tupel und Listen
  - Definition von Funktionen durch rekursive Gleichungen
- Diese Vorlesung: Funktionsdefinition durch
  - Listenkomprehension
  - primitive Rekursion
  - nicht-primitive Rekursion
- Neue Sprachkonzepte:
  - Polymorphie — Erweiterung des Typkonzeptes
  - Lokale Definitionen
- Vordefinierte Funktionen auf Listen

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.

```
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: 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:

```
(++) :: 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

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

Lokale Definitionen

- Lokale Definitionen mit `where` oder `let`:

```
f x y
  | g1 = P
  | otherwise = Q where
    v1 = M
    v2 x = N x
```

```
f x y =
 let v1 = M
     v2 x = N x
 in  if g1 then P
            else Q
```

- `v1`, `v2`, . . . werden gleichzeitig definiert (Rekursion!);
- Namen `v1` und Parameter (`x`) überlagern andere;
- Es gilt die Abseitsregel

  ⇒ Auf gleiche Einrückung der lokalen Definition achten;

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:[]`

Formen der Rekursion auf Listen

- Primitive Rekursion ist Spezialfall der allgemeinen Rekursion
- Primitive Rekursion: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste
- Beispiel:

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

- Weitere Beispiele: `length`, `concat`, `(++)`, . . .

- Auswertung:

$$\begin{array}{lcl} \texttt{sum [4,7,3]} & \leadsto & \texttt{4 + 7 + 3 + 0} \\ \texttt{concat [A, B, C]} & \leadsto & \texttt{A ++ B ++ C++ []} \end{array}$$

- Allgemeines Muster:

  Definition:

$$\begin{array}{lcl} \texttt{f []} & = & \texttt{A} \\ \texttt{f (x:xs)} & = & \texttt{x} \otimes \texttt{f xs} \end{array}$$

  Auswertung:

$$\texttt{f}[\texttt{x1}, ..., \texttt{xn}] = \texttt{x1} \otimes \texttt{x2} \otimes \ldots \otimes \texttt{xn} \otimes \texttt{A}$$

  - Startwert (für die leere Liste) `A :: b`
  - Rekursionsfunktion $\otimes$ `:: a -> b-> b`
- Entspricht einfacher Iteration (`while`-Schleife).
- Vergleiche `Iterator` und `Enumeration` in JAVA.

Nicht-primitive Rekursion

- Allgemeine Rekursion:
  - Rekursion über mehrere Argumente
  - Rekursion über andere Datenstruktur
  - Andere Zerlegung als Kopf und Rest
- Rekursion über mehrere Argumente:

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

- Rekursion über ganzen Zahlen:

```
take :: Int-> [a]-> [a]
take 0 _  = []
take _ [] = []
take n (x:xs)
  | n > 0     = x: take (n-1) xs
  | otherwise = error "take: negative Argument"
```

- Andere Zerlegung — Quicksort:
  - zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
  - sortiere Teilstücke, konkateniere Ergebnisse

- Andere Zerlegung — Mergesort:
  - teile Liste in der Hälfte,
  - sortiere Teilstücke, füge ordnungserhaltend zusammen.

```
msort :: [Int]-> [Int]
msort xs
  | length xs <= 1 = xs
  | otherwise = merge (msort front) (msort back) where
    (front, back) = splitAt ((length xs) `div` 2) xs
    merge :: [Int]-> [Int]-> [Int]
    merge [] x = x
    merge y [] = y
    merge (x:xs) (y:ys)
      | x<= y      = x:(merge xs (y:ys))
      | otherwise = y:(merge (x:xs) ys)
```

Beispiel: das n-Königinnen-Problem

- Problem: n Königinnen auf $n \times n$-Schachbrett
- Spezifikation:
  - Position der Königinnen

    ```
    type Pos = (Int, Int)
    ```
  - Eingabe: Anzahl Königinnen, Rückgabe: Positionen

    ```
    queens :: Int-> [[Pos]]
    ```
- Rekursive Formulierung:
  - Keine Königin— kein Problem.
  - Lösung für n Königinnen: Lösung für $n-1$ Königinnen, n-te Königin so stellen, dass sie keine andere bedroht.
  - n-te Königin muß in n-ter Spalte plaziert werden.

- Hauptfunktion:

```
queens num = putqueens num where
  putqueens :: Int-> [[Pos]]
  putqueens n | n == 0  = [[]]
              | otherwise =
     [ p++ [(n, m)] | p <- putqueens (n-1),
                      m <- [1.. num],
                      safe p (n, m)]
```

  - `[n..m]`: Liste der Zahlen von `n` bis `m`
  - Mehrere Generatoren in Listenkomprehension.
  - Rekursion über Anzahl der Königinnen.

- Sichere neue Position: durch keine andere bedroht

```
safe :: [Pos]-> Pos-> Bool
safe others nu =
   and [ not (threatens other nu)
         | other <- others ]
```

  - Verallgemeinerte Konjunktion `and :: [Bool]-> Bool`

- Gegenseitige Bedrohung:
  - Bedrohung wenn in gleicher Zeile, Spalte, oder Diagonale.

```
threatens :: Pos-> Pos-> Bool
threatens (i, j) (m, n) =
   (j== n) || (i+j == m+n) || (i-j == m-n)
```

  - Test auf gleicher Spalte `i==m` unnötig. Testen.

Zusammenfassung

- Schemata für Funktionen über Listen:
  - Listenkomprehension
  - primitive Rekursion
  - allgemeine Rekursion
- Polymorphie :
  - Abstraktion über Typen durch Typvariablen
- Lokale Definitionen mit `where` und `let`
- Überblick: vordefinierte Funktionen auf Listen