Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 12.11.2008: Typvariablen und Polymorphie

Christoph Lüth

WS 08/09

Fahrplan

- Teil I: Grundlagen
  - Rekursion als Berechnungsmodell
  - Rekursive Datentypen, rekursive Funktionen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung
  - Funktionaler Entwurf, Standarddatentypen
- Teil II: Abstraktion
- Teil III: Beispiele, Anwendungen, Ausblicke

Inhalt

- Letzte Vorlesung: rekursive Datentypen
- Diese Vorlesung:
  - Abstraktion über Typen: Typvariablen und Polymorphie
  - Typinferenz: Wie bestimme ich den Typ eines Ausdrucks?

Letzte Vorlesung: Zeichenketten

- Eine Zeichenkette ist
  - entweder leer (das leere Wort ϵ)
  - oder ein Zeichen und eine weitere Zeichenkette

```
data MyString = Empty
              | Cons Char MyString
```

Funktionen auf Zeichenketten

- Länge:

```
len :: MyString-> Int
len Empty        = 0
len (Cons c str) = 1+ len str
```

- Verkettung:

```
cat :: MyString-> MyString-> MyString
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

- Umkehrung:

```
rev :: MyString-> MyString
rev Empty      = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

Weiteres Beispiel: Liste von Zahlen

- Eine Liste von Zahlen ist
  - entweder leer (das leere Wort ϵ)
  - oder eine Zahl und eine weitere Liste

```
data IntList = Empty
             | Cons Int IntList
```

Funktionen auf Zahlenlisten

- Länge:

```
len :: IntList-> Int
len Empty        = 0
len (Cons c str) = 1+ len str
```

- Verkettung:

```
cat :: IntList-> IntList-> IntList
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

- Umkehrung:

```
rev :: IntList-> IntList
rev Empty      = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

Typvariablen

- Typvariablen abstrahieren über Typen

```
data List a = Empty
            | Cons a (List a)
```

- `a` ist eine Typvariable
- `a` kann mit `Char` oder `Int` instantiiert werden
- `List a` ist ein polymorpher Datentyp
- Typvariable `a` wird bei Anwendung instantiiert
- Signatur der Konstruktoren

```
Empty :: List a
Cons  :: a-> List a-> List a
```

Polymorphe Datentypen

- Typkorrekte Terme:

	Typ
`Empty`	`List a`
`Cons 57 Empty`	`List Int`
`Cons 7 (Cons 8 Empty)`	`List Int`
`Cons 'p' (Cons 'i' (Cons '3' Empty))`	`List Char`
`Cons True Empty`	`List Bool`

- Nicht typ-korrekt:

```
Cons 'a' (Cons 0 Empty)
Cons True (Cons 'x' Empty)
```

wegen Signatur des Konstruktors:

```
Cons :: a-> List a-> List a
```

Polymorphe Funktionen

- Verkettung von `MyString`:

```
cat :: MyString-> MyString-> MyString
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

- Verkettung von `IntList`:

```
cat :: IntList-> IntList-> IntList
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

- Gleiche Definition, unterschiedlicher Typ

  $\rightsquigarrow$ Zwei Instanzen einer allgemeineren Definition.

Polymorphe Funktionen

- Polymorphie erlaubt Parametrisierung über Typen:

```
cat :: List a -> List a -> List a
cat Empty ys          = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

- Typvariable a wird bei Anwendung instantiiert:

```
cat (Cons 3 Empty) (Cons 5 (Cons 57 Empty))
cat (Cons 'p' (Cons 'i' Empty)) (Cons '3' Empty)
```

aber nicht

```
cat (Cons True Empty) (Cons 'a' (Cons 0 Empty))
```

- Typvariable: vergleichbar mit Funktionsparameter

Tupel

- Mehr als eine Typvariable:

```
data Pair a b = Pair a b
```

  - Konstruktorname = Typname

- Beispielterme:

```
Pair 4 "fünf"
Pair (Cons True Empty) 'a'
Pair (3+ 4) (Cons 'a' Empty)
```

Typinferenz

- Bestimmung des Typen durch Typinferenz
- Formalismus: Typableitungen der Form

$$A \vdash x :: t$$

  - A — Typumgebung (Zuordnung Symbole zu Typen)
  - x — Term
  - t — Typ
- Herleitung durch fünf Basisregeln
  - Notation: $t\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right]$ x in t durch s ersetzt
  - Lambda-Abstraktion: $f = \backslash \texttt{x} \texttt{ -> } E$ für $\texttt{f x} = E$

Typinferenzregeln

$$\frac{}{A, x :: t \vdash x :: t}\ Ax$$

$$\frac{A, x :: s \vdash e :: t}{A \vdash \backslash x \texttt{ -> } e :: s \texttt{ -> } t}\ Abs$$

$$\frac{A \vdash e :: s \texttt{ -> } t \quad A \vdash e' :: s}{A \vdash e\ e' :: t}\ App$$

$$\frac{A \vdash e :: t, \text{Typvariable } \alpha \text{ nicht frei in } A}{A \vdash e :: t \left[\begin{smallmatrix} s \\ \alpha \end{smallmatrix}\right]}\ Spec$$

$$\frac{A \vdash f :: s \quad A \vdash c_i :: s \quad A \vdash e_i :: t}{A \vdash \texttt{case } f \texttt{ of } c_i \texttt{ -> } e_i :: t}\ Cases$$

Polymorphie in anderen Programmiersprachen: Java

- Polymorphie in Java: Methode auf alle Subklassen anwendbar

```
class List {
    public List(Object theElement, List n) {
        element = theElement;
        next    = n;  }
    public Object element;
    public List   next; }
```

- Keine Typvariablen:

```
String s = "abc";
List l   = new List(s, null);
```

- `l.element` hat Typ `Object`, nicht `String`

```
String e = (String)l.element;
```

- Neu ab Java 1.5: Generics — damit echte Polymorphie möglich

Polymorphie in anderen Programmiersprachen: C

- "Polymorphie" in C: `void *`

```
struct list {
   void        *head;
   struct list *tail;
   }
```

- Gegeben:

```
int x = 7;
struct list s = { &x, NULL };
```

- `s.head` hat Typ `void *`:

```
int y;
y= *(int *)s.head;
```

- Nicht möglich: `head` direkt als Skalar (e.g. `int`)
- C++: Templates

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter syntaktischer Zucker
- Tupel sind das kartesische Produkt

```
data (a, b) = (a, b)
```

  - `(a, b)` = alle Kombinationen von Werten aus `a` und `b`
  - Auch n-Tupel: `(a,b,c)` etc.

- Listen

```
data [a] = [] | a : [a]
```

  - Weitere Abkürzungen: `[x]= x:[]`, `[x,y] = x:y:[]` etc.

Übersicht: vordefinierte Funktionen auf Listen I

`++`	`[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)]`	Paare zu Liste von Paaren
`unzip`	`[(a, b)]-> ([a], [b])`	Liste von Paaren zu Paaren
`and`, `or`	`[Bool]-> Bool`	Konjunktion/Disjunktion
`sum`	`[Int]-> Int` (überladen)	Summe
`product`	`[Int]-> Int` (überladen)	Produkt

Zeichenketten: `String`

- `String` sind Listen von Zeichen:

```
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)
- 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)
```

- Kritik:
  - Unterschied zwischen Groß- und kleinschreibung

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

- Nichtbuchstaben sollten nicht berücksichtigt werden.

Zusammenfassung

- Typvariablen und Polymorphie: Abstraktion über Typen
- Typinferenz (Hindley-Damas-Milner): Herleitung des Typen eines Ausdrucks
- Vordefinierte Typen: Listen `[a]` und Tupel `(a,b)`
- Nächste Woche: Funktionen höherer Ordnung