Praktische Informatik 3: Funktionale Programmierung
Vorlesung 4 vom 04.11.2014: Typvariablen und Polymorphie

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Rekursive Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
  - Typinferenz
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- Letzte Vorlesung: rekursive Datentypen
- Diese Vorlesung:
  - Abstraktion über Typen: Typvariablen und Polymorphie
  - Arten der Polymorphie:
    - Parametrische Polymorphie
    - Ad-hoc Polymorphie

Ähnliche Datentypen der letzten Vorlesung

```
data Lager = LeeresLager
           | Lager Artikel Menge Lager
```

```
data Einkaufswagen = LeererWagen
                   | Einkauf Artikel Menge Einkaufswagen
```

```
data Path = Cons Id Path
          | Mt
```

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

- ein konstanter Konstruktor
- ein linear rekursiver Konstruktor

Ähnliche Funktionen der letzten Vorlesung

- Pfade:

```
cat :: Path→ Path→ Path
cat Mt q          = q
cat (Cons i p) q = Cons i (cat p q)
```

```
rev :: Path→ Path
rev Mt          = Mt
rev (Cons i p) = cat (rev p) (Cons i Mt)
```

- Zeichenketten:

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

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


Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 0
kasse (Einkauf a m e) = cent a m+ kasse e
```

```
inventur :: Lager→ Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m+ inventur l
```

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

- ein Fall pro Konstruktor
- linearer rekursiver Aufruf

Die Lösung: Polymorphie

Definition (Polymorphie)

Polymorphie ist Abstraktion über Typen

Arten der Polymorphie

- Parametrische Polymorphie (Typvariablen):
  Generisch über alle Typen
- Ad-Hoc Polymorphie (Überladung):
  Nur für bestimmte Typen

Anders als in Java (mehr dazu später).

Parametrische Polymorphie: Typvariablen

- Typvariablen abstrahieren über Typen

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

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

```
Empty :: List α
Cons  :: α→ List α→ List α
```

Polymorphe Ausdrücke

- Typkorrekte Terme:

Empty	List α
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 :: α→ List α→ List α
```


Polymorphe Funktionen

- Parametrische Polymorphie für Funktionen:

```
cat :: List α → List α → List α
cat Empty ys          = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

- Typvariable α 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
- Restriktion: Typvariable auf Resultatposition?

Beispiel: Der Shop (refaktoriert)

- Einkaufswagen und Lager als Listen?
- Problem: zwei Typen als Argument
- Lösung 1: zu einem Typ zusammenfassen

```
data Posten = Posten Artikel Menge
```

- Damit:

```
type Lager = [Posten]
```

```
type Einkaufswagen = [Posten]
```

- Gleicher Typ!
  - Bug or Feature? Bug!
- Lösung: Datentyp verkapseln

```
data Lager = Lager [Posten]
data Einkaufswagen = Einkaufswagen [Posten]
```


Lösung 2: Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

- Beispielterm / Typ

Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	Pair (List Bool) Char
Pair (3+ 4) (Cons 'a' Empty)	Pair Int (List Char)
Cons (Pair 7 'x') Empty	List (Pair Int Char)

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter syntaktischer Zucker
- Tupel sind das kartesische Produkt

```
data (α, β) = (α, β)
```

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

- Listen

```
data [α] = [] | α : [α]
```

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

Vordefinierte Datentypen: Optionen

```
data Preis = Cent Int | Ungueltig
```

```
data Resultat = Gefunden Menge | NichtGefunden
```

```
data Trav = Succ Path
          | Fail
```

Instanzen eines vordefinierten Typen:

```
data Maybe α = Just α | Nothing
```

Vordefinierten Funktionen (**import** Data.Maybe):

```
fromJust   :: Maybe α → α
fromMaybe  :: α→ Maybe α→ α
maybeToList :: Maybe α→ [α]
listToMaybe :: [α]→ Maybe α  — "sicheres" head
```


Übersicht: vordefinierte Funktionen auf Listen I

```
(++)    :: [α]→ [α]→ [α]     — Verketten
(!!)    :: [α]→ Int→ α       — n-tes Element selektieren
concat  :: [[α]]→ [α]        — "flachklopfen"
length  :: [α]→ Int          — Länge
head, last :: [α]→ α         — Erstes/letztes Element
tail, init :: [α]→ [α]       — Hinterer/vorderer Rest
replicate  :: Int→ α→ [α]    — Erzeuge n Kopien
repeat  :: α→ [α]            — Erzeugt zyklische Liste
take    :: Int→ [α]→ [α]     — Erste n Elemente
drop    :: Int→ [α]→ [α]     — Rest nach n Elementen
splitAt :: Int→ [α]→ ([α], [α])— Spaltet an Index n
reverse :: [α]→ [α]          — Dreht Liste um
zip     :: [α]→ [β]→ [(α, β)] — Erzeugt Liste v. Paaren
unzip   :: [(α, β)]→ ([α], [β]) — Spaltet Liste v. Paaren
and, or :: [Bool]→ Bool      — Konjunktion/Disjunktion
sum     :: [Int]→ Int        — Summe (überladen)
```


Vordefinierte Datentypen: Zeichenketten

- String sind Listen von Zeichen:

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

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

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

- Beispiel:

```
cnt :: Char→ String→ Int
cnt c []     = 0
cnt c (x:xs) = if (c== x) then 1+ cnt c xs
               else cnt c xs
```

Zurück im Labyrinth

- Labyrinth als Instanz eines allgemeineren Datentyps?
- Erste Refaktorierung:

```
type Id   = Integer
```

```
type Path = [Id]
```

```
data Lab = Node Id [Lab]
```

- Instanz eines variadischen Baumes

Labyrinth verallgemeinert: Variadische Bäume

- Variable Anzahl Kinderknoten: Liste von Kinderknoten

```
data VTree α = VNode α [VTree α]
```

- Anzahl Knoten zählen:

```
count :: VTree α→ Int
count (VNode _ ns) = 1+ count_nodes ns
```

```
count_nodes :: [VTree α] → Int
count_nodes []     = 0
count_nodes (t:ts) = count t+ count_nodes ts
```

- Damit: das Labyrinth als variadischer Baum

```
type Lab = VTree Id
```


Ad-Hoc Polymorphie und Overloading

Definition (Überladung)
Funktion f :: a→ b existiert für mehr als einen, aber nicht für alle Typen

- Beispiel:
  - Gleichheit: (==) :: α→ α→ Bool
  - Vergleich: (<) :: α→ α→ Bool
  - Anzeige: show :: α→ String
- Lösung: Typklassen
- Typklassen bestehen aus:
  - Deklaration der Typklasse
  - Instantiierung für bestimmte Typen

Typklassen: Syntax

- Deklaration:

```
class Show α where
  show :: α→ String
```

- Instantiierung:

```
instance Show Bool where
  show True  = "Wahr"
  show False = "Falsch"
```

- Prominente vordefinierte Typklassen
  - Eq für (==)
  - Ord für (<) (und andere Vergleiche)
  - Show für show
  - Num (uvm) für numerische Operationen (Literale überladen)

Typklassen in polymorphen Funktionen

- Element einer Liste (vordefiniert):

```
elem :: Eq α ⇒ α→ [α]→ Bool
elem e []     = False
elem e (x:xs) = e == x || elem e xs
```

- Sortierung einer List: qsort

```
qsort :: Ord α ⇒ [α]→ [α]
```

- Liste ordnen und anzeigen:

```
showsorted :: (Eq α, Show α)⇒ [α]→ String
showsorted x = show (qsort x)
```


Hierarchien von Typklassen

- Typklassen können andere voraussetzen:

```
class Eq α⇒ Ord α where
  (<)  :: α→ α→ Bool
  (≤)  :: α→ α→ Bool
  a ≤ b = a == b || a < b
```

- Default-Definition von ≤
- Kann bei Instantiierung überschrieben werden

Polymorphie: the missing link

	Parametrisch	Ad-Hoc
Funktionen	f :: α→ Int	**class** F α **where** f :: a→ Int
Typen	**data** Maybe α= Just α | Nothing	Konstruktorklassen

- Kann Entscheidbarkeit der Typherleitung gefährden
- Erstmal nicht relevant

Polymorphie in anderen Programmiersprachen: Java

- Polymorphie in Java: Methode auf alle Subklassen anwendbar
  - Manuelle Typkonversion nötig, fehleranfällig
- Neu ab Java 1.5: Generics
  - Damit parametrische Polymorphie möglich

```
class AbsList<T> {
  public AbsList(T el, AbsList<T> tl) {
    this.elem= el;
    this.next= tl;
  }
  public T elem;
  public AbsList<T> next;
```

Polymorphie in anderen Programmiersprachen: Java

Typkorrekte Konkatenenation:

```
void concat(AbsList<T> o)
{
  AbsList<T> cur= this;
  while (cur.next != null) cur= cur.next;
  cur.next= o;
}
```

Nachteil: Benutzung umständlich, weil keine Typherleitung

```
AbsList<Integer> l=
  new AbsList<Integer>(new Integer(1),
    new AbsList<Integer>(new Integer(2), null));
```


Polymorphie in anderen Programmiersprachen: Java

- Ad-Hoc Polymorphie: Interface und abstrakte Klassen
- Flexibler in Java: beliebige Parameter etc.

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

Zusammenfassung

- Abstraktion über Typen
  - Uniforme Abstraktion: Typvariable, parametrische Polymorphie
  - Fallbasierte Abstraktion: Überladung, ad-hoc-Polymorphie
- In der Sprache:
  - Typklassen
  - polymorphe Funktionen und Datentypen
- Vordefinierte Typen: Listen [a] und Tupel (a,b)
- Nächste Woche: Abstraktion über Funktionen

$\rightsquigarrow$ Funktionen höherer Ordnung