

Praktische Informatik 3: Funktionale Programmierung Vorlesung 4 vom 06.11.2012: Typvariablen und Polymorphie

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Rev. 1861

1 [28]

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

2 [28]

Inhalt

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

3 [28]

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

4 [28]

Ähnliche Funktionen der letzten Vorlesung

- ▶ Pfade:

```
cat :: Path → Path → Path
cat Mt p = p
cat (Cons p ps) qs = Cons p (cat ps qs)
```

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

5 [28]

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

6 [28]

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

7 [28]

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
```

8 [28]

Polymorphe Ausdrücke

- **Typkorrekte** Terme:
 - Empty Typ
 - Cons 57 Empty List α
 - 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 ::  $\alpha \rightarrow$  List  $\alpha \rightarrow$  List  $\alpha$ 
```

9 [28]

Polymorphe Funktionen

- Parametrische Polymorphie für Funktionen:

```
cat :: List  $\alpha \rightarrow$  List  $\alpha \rightarrow$  List  $\alpha$ 
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?

10 [28]

Beispiel: Der Shop (refaktoriert)

- Einkaufswagen und Lager als Listen:

```
type Lager = [(Artikel, Menge)]
```

```
type Einkaufswagen = [(Artikel, Menge)]
```

- Gleicher Typ!

- Bug or Feature?

- Lösung: Datentyp verkapseln

```
data Lager = Lager [(Artikel, Menge)]
    deriving (Eq, Show)
data Einkaufswagen = Einkaufswagen [(Artikel, Menge)]
    deriving (Eq, Show)
```

11 [28]

Tupel

- Mehr als eine Typvariable möglich

- Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair  $\alpha \beta$  = Pair  $\alpha \beta$ 
```

- Signatur des Konstruktors:

```
Pair ::  $\alpha \rightarrow \beta \rightarrow$  Pair  $\alpha \beta$ 
```

- 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)

12 [28]

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter **syntaktischer Zucker**

- **Tupel** sind das kartesische Produkt

```
data ( $\alpha$ ,  $\beta$ ) = ( $\alpha$ ,  $\beta$ )
```

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

- **Listen**

```
data [ $\alpha$ ] = [] |  $\alpha$  : [ $\alpha$ ]
```

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

13 [28]

Vordefinierte Datentypen: Optionen

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

```
data Resultat = Gefunden Menge | Nichtgefunden
```

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

Instanzen eines **vordefinierten** Typen:

```
data Maybe  $\alpha$  = Just  $\alpha$  | Nothing
```

Vordefinierten Funktionen (import Data.Maybe):

```
fromJust :: Maybe  $\alpha \rightarrow \alpha$ 
fromMaybe ::  $\alpha \rightarrow$  Maybe  $\alpha \rightarrow \alpha$ 
maybeToList :: Maybe  $\alpha \rightarrow$  [ $\alpha$ ]
listToMaybe :: [ $\alpha$ ]  $\rightarrow$  Maybe  $\alpha$  — "sicheres" head
```

14 [28]

Übersicht: vordefinierte Funktionen auf Listen I

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

15 [28]

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  $\rightarrow$  String  $\rightarrow$  Int
cnt c [] = 0
cnt c (x:xs) = if (c==x) then 1+ cnt c xs
               else cnt c xs
```

16 [28]

Zurück im Labyrinth

- Labyrinth als Instanz eines allgemeineren Datentyps?

- Ja: **variadische** Bäume

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

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

```
type Trav = Maybe Path
```

17 [28]

Labyrinth verallgemeinert: Variadische Bäume

- Variable Anzahl Kinderknoten: Liste von Kinderknoten

```
data VTree α = Node 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
```

18 [28]

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)

Funktion $f :: a \rightarrow b$ existiert für **mehr als einen**, aber **nicht für alle** Typen

- Beispiel:

- Gleichheit: $(==) :: a \rightarrow a \rightarrow \text{Bool}$
- Vergleich: $(<) :: a \rightarrow a \rightarrow \text{Bool}$
- Anzeige: $\text{show} :: a \rightarrow \text{String}$

- Lösung: **Typklassen**

- Typklassen bestehen aus:

- **Deklaration** der Typklasse
- **Instanziierung** für bestimmte Typen

19 [28]

Typklassen: Syntax

- **Deklaration:**

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

- **Instanziierung:**

```
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 (Literele überladen)

20 [28]

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)
```

21 [28]

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 \leq

- Kann bei Instanziierung überschrieben werden

22 [28]

Polymorphie: the missing link

	Parametrisch	Ad-Hoc
Funktionen	$f :: \alpha \rightarrow \text{Int}$	class F α where $f :: a \rightarrow \text{Int}$
Typen	data Maybe α = Just α Nothing	Konstruktorklassen

- Kann **Entscheidbarkeit** der Typherleitung gefährden

- Erstmal nicht relevant

23 [28]

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;
}
```

24 [28]

Polymorphie in anderen Programmiersprachen: Java

Typkorrekte Konkatenation:

```
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));
```

25 [28]

Polymorphie in anderen Programmiersprachen: Java

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

26 [28]

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

27 [28]

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

~> Funktionen höherer Ordnung

28 [28]