Praktische Informatik 3: Funktionale Programmierung
Vorlesung 4 vom 06.11.2018: Typvariablen und Polymorphie

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Fahrplan

- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Zyklische Datenstrukturen
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- Letzte Vorlesungen: algebraische 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 String = Empty
            | Char :+ String
```

- ein **konstanter** Konstruktor
- ein **linear rekursiver** Konstruktor

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
```

```
length :: String→ Int
length Empty     = 0
length (c :+ s) = 1+ length s
```

- 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

Parametrische Polymorphie: Typvariablen

- **Typvariablen** abstrahieren über Typen

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

- α ist eine **Typvariable**
- List α ist ein **polymorpher** Datentyp
- Signatur der Konstruktoren

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

- Typvariable α wird bei Anwendung instantiiert

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

	Typ
`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**:

```
(++) :: List α → List α → List α
Empty ++ t       = t
(Cons c s) ++ t = Cons c (s++ t)
```

  - Typvariable vergleichbar mit Funktionsparameter

- Typvariable α wird bei Anwendung instantiiert:

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

aber **nicht**

```
Cons True Empty ++ Cons 'a' (Cons 'b' Empty)
```


Beispiel: Der Shop (refaktoriert)

- Einkaufswagen und Lager als Listen?
- Problem: zwei Typen als Argument
- Lösung: 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 = Ekwg [Posten]
```


Tupel

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

```
data Pair α β = Pair { left :: α, right :: β }
```

- Signatur Konstruktor und Selektoren:

```
Pair  :: α → β → Pair α β
left  :: Pair α β → α
right :: Pair α β → β
```

- Beispielterm

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

Vordefinierte Datentypen

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter **syntaktischer Zucker**
- **Listen**

```
data [α] = [] | α : [α]
```

  - Weitere Abkürzungen:
    Listenliterale: `[x]` für `x:[]`, `[x,y]` für `x:y:[]` etc.
    Aufzählungen: `[n .. m]` und `[n, m .. k]` für aufzählbare Typen

- **Tupel** sind das kartesische Produkt

```
data (α, β) = ( fst :: α, snd :: β)
```

  - `(a, b)` = alle Kombinationen von Werten aus a und b
  - Auch n-Tupel: `(a,b,c)` etc. (aber ohne Selektoren)
  - 0-Tupel: `()` (*unit type*, Typ mit genau einem Element)

Vordefinierte Datentypen: Optionen

- Existierende Typen:

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

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

- Instanzen eines **vordefinierten** Typen:

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

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

```
fromJust    :: Maybe α → α      — partiell
fromMaybe   :: α → Maybe α → α
listToMaybe :: [α] → Maybe α    — totale Variante von head
maybeToList :: Maybe α → [α]    — rechtsinvers zu listToMaybe
```

- Es gilt: `listToMaybe (maybeToList m) = m`
  `length l ≤ 1 ⟹ maybeToList (listToMaybe l) = l`

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 von Paaren
unzip   :: [(α, β)] → ([α], [β]) — Spaltet Liste von 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** für Stringliterale:

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

- Beispiele:

```
"abc" !! 1  ⇝  'b'
reverse "oof"  ⇝  "foo"
['a','c'..'z']  ⇝  "acegikmoqsuwy"
splitAt 10 "Praktische␣Informatik"  ⇝
           ("Praktische","␣Informatik")
```


Typherleitung

Typen in Haskell (The Story So Far)

- Primitive Basisdatentypen: Bool, Double
- Funktionstypen: Double→ Int→ Int, [Char] → Double
- Typkonstruktoren: [], (...), Foo
- Typvariablen
  fst :: (α, β) → α
  length :: [α] → Int
  (++) :: [α] → [α] → [α]
- Typklassen :
  elem :: Eq a⇒ a → [a] → Bool
  max :: Ord a⇒ a → a → a

Typinferenz: Das Problem

- Gegeben Definition von f:

```
f m xs = m + length xs
```

- Frage: welchen Typ hat f?
  - Unterfrage: ist die angegebene Typsignatur korrekt?
- **Informelle** Ableitung

```
f m xs  =  m  +  length  xs
                 [α]→ Int
                          [α]
                      Int
            Int
               Int

f :: Int→ [α]→ Int
```


Typinferenz (nach Hindley-Milner)

- Typinferenz: **Herleitung** des Typen eines Ausdrucks
- Für bekannte Bezeichner wird Typ eingesetzt
- Für Variablen wird allgemeinster Typ angenommen
- Bei der Funktionsanwendung wird **unifiziert**:

```
f m xs  =  m      +      length   xs

           α             [β]→ Int  γ
                                  [β]  γ ↦ [β]
                              Int
               Int→ Int→ Int
           Int                          α ↦ Int
               Int→ Int
                          Int

 f :: Int→ [β]→ Int
```


Typinferenz

Theorem (Entscheidbarkeit der Typinferenz)
Die Typinferenz ist ***entscheidbar****, und findet immer den* ***allgemeinsten*** *Typ, wenn er existiert.*

- Entscheidbarkeit ist nicht alles.
- Grundsätzliche Komplexität ist *DEXPTIME*(n) (deterministisch exponentiell), aber in der Praxis ist das **nie** ein Problem.

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

```
f x y =  (x, 3)    :   ('f', y)   :   []
          α  Int        Char  β        [γ]
         (α, Int)      (Char, β)
                           [(Char, β)]     γ ↦ (Char, β)
             [(Char, Int)]                 β ↦ Int, α ↦ Char
  f :: Char→ Int→ [(Char, Int)]
```

- Allgemeinster Typ **muss nicht** existieren (Typfehler!)
  - Bsp: [True] ++ [3] , x : x

Ad-Hoc Polymorphie

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 :: (Ord 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

Abschließende Bemerkungen

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

Zusammenfassung

- **Abstraktion** über Typen
  - Uniforme Abstraktion: Typvariable, parametrische Polymorphie
  - Fallbasierte Abstraktion: Überladung, ad-hoc-Polymorphie
- In der Sprache Haskell: **Typvariablen** und **Typklassen**
- Wichtige **vordefinierte** Typen:
  - Listen [α]
  - Optionen Maybe α
  - Tupel (α,β)