Praktische Informatik 3: Funktionale Programmierung

Vorlesung 4 (08.11.2022): Typvariablen und Polymorphie

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Fahrplan

Inhalt

- Letzte Vorlesungen: algebraische Datentypen
- Diese Vorlesung:
  - **Abstraktion** über Typen: Typvariablen und Polymorphie
  - Arten der Polymorphie:
    - Parametrische Polymorphie
    - Ad-hoc Polymorphie
  - Typableitung in Haskell

Lernziele

Wir verstehen, wie in Haskell die Typableitung funktioniert, und was Signaturen wie `head :: [α] → α` und `elem :: Eq α ⇒ α → [α] → Bool` bedeuten.

Ähnliche Datentypen der letzten Vorlesung

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

```
data Einkaufskorb = LeererKorb
                  | Einkauf Artikel Menge Einkaufskorb
```

```
data MyString = Empty
              | Char :+ MyString
```

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

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufskorb→ Int
kasse LeererKorb = 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 :: MyString→ 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).

I. 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:

Term	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 'p' (Cons 'i' Empty) ++ Cons '3' Empty
```

```
Cons 3 Empty ++ Cons 5 (Cons 57 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

```
type Lager = List (Artikel Menge)
```

  - Geht so **nicht!**
- Lösung: zu einem Typ zusammenfassen

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

- Damit:

```
type Lager = List Posten
type Einkaufskorb = List Posten
```

- **Gleicher** Typ!

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)`

☞ Siehe Übung 4.1

II. 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]              — Verkettet zwei Listen
(!!)          :: [α]→ Int→ α                — n-tes Element selektieren, gezählt ab 0
concat        :: [[α]]→ [α]                 — "flachklopfen"
length        :: [α]→ Int                   — Länge
head, last    :: [α]→ α                     — Erstes bzw. letztes Element
tail, init    :: [α]→ [α]                   — Hinterer bzw. vorderer Rest
replicate     :: Int→ α→ [α]                — Erzeuge n Kopien
repeat        :: α→ [α]                     — Erzeugt zyklische Liste
take, drop    :: Int→ [α]→ [α]              — Erste bzw. letzte n Elemente
splitAt       :: Int→ [α]→ ([α], [α])       — Spaltet an Index n, gezählt ab 0
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, product  :: [Int]→ Int                 — Summe und Produkt (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")
```

☞ Siehe Übung 4.2

III. Ad-Hoc Polymorphie

Parametrische Polymorphie: Grenzen

- Eine Funktion `f`: $\alpha \to \beta$ funktioniert auf **allen** Typen **gleich**.
- Nicht immer der Fall:
  - Gleichheit: `(==) :: α→ α→ Bool`

    Nicht auf allen Typen ist Gleichheit entscheidbar (besonders **Funktionen**)
  - Ordnung: `(≤) :: α→ α→ Bool`

    Nicht auf allen Typen definiert
  - Anzeige: `show :: α→ String`

    Konversion in Zeichenketten höchst divers (Zeichenketten, Listen, Zahlen. . .)

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)
Funktion `f :: α→ β` existiert für **mehr als einen**, aber **nicht** für **alle** Typen

- Lösung: **Typklassen**
- Typklassen bestehen aus:
  - **Deklaration** der Typklasse
  - **Instantiierung** für bestimmte Typen
- **Achtung**: hat wenig mit Klassen in Java zu tun

Typklassen: Syntax

- **Deklaration**:

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

- **Instantiierung**:

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


Prominente vordefinierte Typklassen

- Gleichheit: `Eq` für `(==)`
- Ordnung: `Ord` für `(≤)` (und andere Vergleiche)
- Anzeigen: `Show` für `show`
- Lesen: `Read` für `read :: String→ α` (Achtung: Laufzeitefehler!)
- Numerische Typklassen:
  - `Num` für `0, 1, +, -`
  - `Integral` für `quot, rem, div, mod`
  - `Fractional` für `/`
  - `Floating` für `exp, log, sin, cos`

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

☞ Siehe Übung 4.3

IV. Typherleitung

Typen in Haskell (The Story So Far)

- Primitive Basisdatentypen: `Bool`, `Double`
- Funktionstypen `Double` → `Int` → `Int`, `[Char]` → `Double`
- Typkonstruktoren: `[]`, `(...)`, `Foo`
- Typvariablen
  - `fst` :: $(\alpha, \beta) \to \alpha$
  - `length` :: $[\alpha] \to$ `Int`
  - $(+\!\!+)$ :: $[\alpha] \to [\alpha] \to [\alpha]$
- Typklassen :
  - `elem` :: `Eq` $\alpha \Rightarrow \alpha \to [\alpha] \to$ `Bool`
  - `max` :: `Ord` $\alpha \Rightarrow \alpha \to \alpha \to \alpha$

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.

DEMO

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!)

Und was ist mit Typklassen?

- Typklassen schränken den Typ ein
- Typklassen werden bei der Unifikation **vereinigt**:

```
  elem                          3
Eq α :: α→[α]→ Bool        Num β :: β
              elem 3
        (Eq α, Num α) :: [α]→ Bool
```

- Instantiierung muss Typklassen berücksichtigen:

```
     elem  3                    "abc"
(Eq α, Num α) :: [α]→ Bool      [Char]       α |→ Char
```

  - `Char` muss Instanz von `Eq` und `Num` sein.

Typfehler

- Typfehler treten auf, wenn zwei Typen t_1, t_2 nicht **unifiziert** werden können.
- Es gibt drei Arten von Typfehlern:
  1. Typkonstanten nicht unifizierbar: `[True] ++ "a"`
  2. Typ nicht Instanz der geforderten Klasse: `3 + 'a'`
  3. Unifikation gibt **unendlichen** Typ: `x : x`

DEMO

V. Andere Programmiersprachen

Polymorphie in C

- Polymorphie in C: `void *`
- Pointer-to-void ist kompatibel mit allen anderen Pointer-Typen.
- Manueller Typ-Cast nötig
  - Vergl. `Object` in Java
- Extrem Fehleranfälig

Polymorphie in Java

- Polymorphie in **Java**: Methode auf alle Subklassen anwendbar
  - Manuelle Typkonversion nötig, fehleranfällig
- Neu ab Java 1.5: **Generics**
  - Damit **parametrische Polymorphie** möglich
  - **Nachteil:** Benutzung umständlich, weil keine Typherleitung (wegen Kombination mit Subtyping)
  - **Vorteil:** Typkorrektheit sichergestellt
  - Allerdings: Typ-Parameter nur für Klassen, Instanzen nur Objekte.

Ad-Hoc Polymorphie in Java

- `interface` und `abstract class`
- Flexibler in Java: beliebig viele Parameter etc.
- Eingeschränkt durch Vererbungshierarchie
- Ähnliche Standardklassen
  - `toString`
  - `equals` und `==`, keine abgeleitete strukturelle Gleichheit

Polymorphie in Pyton

- In Python werden Typen zur **Laufzeit** geprüft (**dynamic typing**)
- **duck typing**: strukturell gleiche Typen sind gleich
- Polymorphie durch Klassen
- Statt Interfaces kennt Python **Mixins**
  - Abstrakte Klassen ohne Oberklasse

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 $[\alpha]$
  - Optionen `Maybe` α
  - Tupel (α, β)