Praktische Informatik 3: Funktionale Programmierung
Vorlesung 4 vom 08.11.2016: Typvariablen und Polymorphie

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

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

Organisatorisches

- Abgabe der Übungsblätter: Freitag 12 Uhr mittags
- Mittwoch, 09.11.16: Tag der Lehre
  - Tutorium Mi 14-16 (Alexander) verlegt auf Do 14-16 Cartesium 0.01
  - Alle anderen Tutorien finden statt.
- Hinweis: Quellcode der Vorlesung auf der Webseite verfügbar.

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 Path = Cons Id Path
          | Mt
```

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

- 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
```

```
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

Parametrische Polymorphie: Typvariablen

- Typvariablen abstrahieren über Typen

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

- α ist eine Typvariable
- α kann mit Int 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: Typ

  Empty

Polymorphe Ausdrücke

- Typkorrekte Terme:

	Typ
Empty	List α

Polymorphe Ausdrücke

- Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	

Polymorphe Ausdrücke

- Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int

Polymorphe Ausdrücke

- Typkorrekte Terme:

	Typ
`Empty`	`List` α
`Cons 57 Empty`	`List Int`
`Cons 7 (Cons 8 Empty)`	

Polymorphe Ausdrücke

- Typkorrekte Terme:

	Typ
`Empty`	`List` α
`Cons 57 Empty`	`List Int`
`Cons 7 (Cons 8 Empty)`	`List Int`

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))`	

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`

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`	

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` :: $\alpha \rightarrow$ `List` $\alpha \rightarrow$ `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

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?

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 α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

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'
```

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

  `Pair 4 'x'`

  Typ

  `Pair Int Char`

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

  Pair 4 'x'

  Pair (Cons True Empty) 'a'

  Typ

  Pair Int Char

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`

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

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

Typ

```
Pair Int Char
Pair (List Bool) Char
```

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)`

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`	

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

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter syntaktischer Zucker
- Tupel sind das kartesisches Produkt

```
data (α, β) = (α, β)
```

  - (a, b) = alle Kombinationen von Werten aus a und b
  - Auch n-Tupel: (a,b,c) etc. (für $n \leq 9$)

- 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 = 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
```

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]
```

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
```

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

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)

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

- Beispiel:
  - Gleichheit: $(==)$ :: $\alpha \rightarrow \alpha \rightarrow$ Bool
  - Vergleich: $(<)$ :: $\alpha \rightarrow \alpha \rightarrow$ Bool
  - Anzeige: show :: $\alpha \rightarrow$ 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 $(\leq)$

- Kann bei Instantiierung überschrieben werden

Typherleitung

Typen in Haskell (The Story So Far)

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

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

```
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
```

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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

$$\begin{array}{lllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length\ xs} \\ & & & & [\alpha] \rightarrow \mathsf{Int} \end{array}$$

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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

$$\begin{array}{lllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length\ \ xs} \\ & & & & [\alpha] \to \mathsf{Int} \\ & & & & \quad [\alpha] \end{array}$$

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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

$$\begin{array}{lllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length\ \ xs} \\ & & & & [\alpha] \to \mathsf{Int} \\ & & & & \qquad\qquad [\alpha] \\ & & & & \mathsf{Int} \end{array}$$

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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

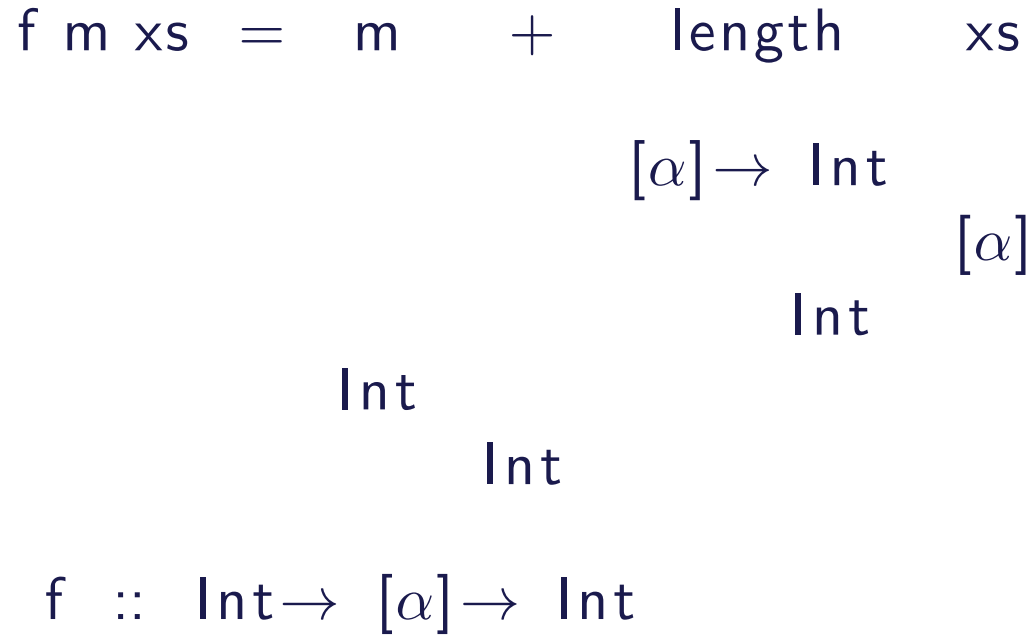
$$\begin{array}{lllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} \quad \mathsf{xs} \\ & & & & [\alpha] \to \mathsf{Int} \\ & & & & \qquad\qquad [\alpha] \\ & & & & \qquad\quad \mathsf{Int} \\ & & \mathsf{Int} & & \end{array}$$

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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



Typinferenz

- 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
```

Typinferenz

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

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \end{array}$$

Typinferenz

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

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \\ & & & & [\beta] & \gamma \mapsto \beta \end{array}$$

Typinferenz

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

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \\ & & & & & [\beta] \quad \gamma \mapsto \beta \\ & & & & \mathsf{Int} & \end{array}$$

Typinferenz

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

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \\ & & & & & [\beta] \quad \gamma \mapsto \beta \\ & & & & \mathsf{Int} & \\ & & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} & & & \end{array}$$

Typinferenz

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

$$
\begin{array}{lllll}
\mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length\ xs} \\
 & & \alpha & & [\beta] \to \mathsf{Int} \quad \gamma \\
 & & & & [\beta] \quad \gamma \mapsto \beta \\
 & & & & \mathsf{Int} \\
 & & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} & & \\
 & & \mathsf{Int} & & \alpha \mapsto \mathsf{Int}
\end{array}
$$

Typinferenz

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

$$
\begin{array}{llllll}
\mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\
 & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \\
 & & & & & [\beta] \quad \gamma \mapsto \beta \\
 & & & & \mathsf{Int} & \\
 & & & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} & & \\
 & & \mathsf{Int} & & & \alpha \mapsto \mathsf{Int} \\
 & & & \mathsf{Int} \to \mathsf{Int} & &
\end{array}
$$

Typinferenz

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

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta]\to\ \mathsf{Int} & \gamma \\ & & & & & [\beta] \quad \gamma \mapsto \beta \\ & & & & \mathsf{Int} & \\ & & \mathsf{Int}\to\ \mathsf{Int}\to\ \mathsf{Int} & & & \\ & & \mathsf{Int} & & & \alpha \mapsto \mathsf{Int} \\ & & \mathsf{Int}\to\ \mathsf{Int} & & & \\ & & & & \mathsf{Int} & \end{array}$$

$$\mathsf{f\ ::\ Int}\to\ [\alpha]\to\ \mathsf{Int}$$

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

```
(x, 3)   :   ('f', y)   :        []
```

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

(x,	3)	: (’f’,	y)	: []
α	Int	Char	β	$[\gamma]$

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

$$
\begin{array}{ccccc}
(\mathsf{x},\ 3) & : & (\text{'f'},\ \mathsf{y}) & : & [\,] \\
\alpha \quad \mathsf{Int} & & \mathsf{Char} \quad \beta & & [\gamma] \\
(\alpha,\ \mathsf{Int}) & & (\mathsf{Char},\ \beta) & &
\end{array}
$$

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

$$
\begin{array}{lcll}
(\mathsf{x},\ 3) & : & (\text{'f'},\ \mathsf{y}) & : \quad [\,] \\
\alpha \quad \mathsf{Int} & & \mathsf{Char} \quad \beta & \quad [\gamma] \\
(\alpha,\ \mathsf{Int}) & & (\mathsf{Char},\ \beta) & \\
 & & (\mathsf{Char},\ \beta) \quad [(\mathsf{Char},\ \beta)] & \gamma \mapsto (\mathsf{Char},\ \beta) \\
 & & [(\mathsf{Char},\ \beta)] & \beta \mapsto \mathsf{Int}, \\
 & & & \alpha \mapsto \mathsf{Char}
\end{array}
$$

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

$$
\begin{array}{ccccc}
(x,\ 3) & : & ('f',\ y) & : & [\,] \\
\alpha \quad \mathsf{Int} & & \mathsf{Char} \quad \beta & & [\gamma] \\
(\alpha,\ \mathsf{Int}) & & (\mathsf{Char},\ \beta) & & \\
 & & (\mathsf{Char},\ \beta) & [(\mathsf{Char},\ \beta)] & \gamma \mapsto (\mathsf{Char},\ \beta) \\
 & & [(\mathsf{Char},\ \beta)] & & \beta \mapsto \mathsf{Int}, \\
 & & & & \alpha \mapsto \mathsf{Char} \\
 & [(\mathsf{Char},\ \mathsf{Int})] & & &
\end{array}
$$

- Allgemeinster Typ muss nicht existieren (Typfehler!)

Abschließende Bemerkungen

Polymorphie: the missing link

	Parametrisch	Ad-Hoc
Funktionen	f :: $\alpha \rightarrow$ Int	**class** F α **where** f :: a $\rightarrow$ Int
Typen	**data** Maybe α = Just α | Nothing	

Polymorphie: the missing link

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

- Kann Entscheidbarkeit der Typherleitung gefährden

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

- Ad-Hoc Polymorphie in Java:
  - Interface und abstrakte Klassen
  - Flexibler in Java: beliebige Parameter etc.
- Dynamische Typisierung: Ruby, Python
  - “Duck typing”: strukturell gleiche Typen sind gleich

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]`, Option `Maybe` α und Tupel `(a,b)`
- Nächste Woche: Abstraktion über Funktionen

$\longrightarrow$ Funktionen höherer Ordnung