Praktische Informatik 3: Funktionale Programmierung

Vorlesung 4 vom 23.11.2020: Typvariablen und Polymorphie

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2020/21

Fahrplan

- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Rekursive und zyklische Datenstrukturen
  - 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
  - Typableitung in Haskell

Lernziele

Wir verstehen, wie in Haskell die Typableitung funktioniert, und was Signaturen wie `head` $:: [\alpha] \to \alpha$ und `elem` $::$ `Eq` $\alpha \Rightarrow \alpha \to [\alpha] \to$ `Bool` bedeuten.

Ähnliche Datentypen der letzten Vorlesung

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

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

```haskell
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: Typ

  `Empty`

Polymorphe Ausdrücke

- **Typkorrekte** Terme: Typ

Term	Typ
`Empty`	`List` α

Polymorphe Ausdrücke

▶ **Typkorrekte** Terme:	Typ
`Empty`	`List` α
`Cons 57 Empty`	

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

Term	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:

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

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

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

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`

Polymorphe Ausdrücke

- **Typkorrekte** Terme: Typ

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`	

Polymorphe Ausdrücke

- Typkorrekte Terme:

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**:

```
(++) :: 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)

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

- Signatur Konstruktor und Selektoren:

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

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'
```

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

Beispielterm	Typ
`Pair 4 'x'`	`Pair Int Char`

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

Beispielterm	Typ
`Pair 4 'x'`	`Pair Int Char`
`Pair (Cons True Empty) 'a'`	

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

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 { 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

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

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

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)`

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

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`	

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

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)`

Jetzt seit ihr dran!

Übung 4.1: Neue Typen

Sind folgende Ausdrücke typkorrekt, und wenn ja welchen Typ haben sie?

1. `right (Pair (3 + 4) Empty)`
2. `head (Pair (Cons 'x' Empty) True)`
3. `right (head (Cons (Pair 'x' 3) Empty))`
4. `head (tail (Cons 3 (Cons 4 Empty)))`

Jetzt seit ihr dran!

Übung 4.1: Neue Typen

Sind folgende Ausdrücke typkorrekt, und wenn ja welchen Typ haben sie?

1. `right (Pair (3 + 4) Empty)`
2. `head (Pair (Cons 'x' Empty) True)`
3. `right (head (Cons (Pair 'x' 3) Empty))`
4. `head (tail (Cons 3 (Cons 4 Empty)))`

Lösung:

1. Typ: `List` α
2. Typfehler
3. Typ: `Integer`
4. Typ: `Integer`

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

`(++)`	::	$[\alpha] \to [\alpha] \to [\alpha]$	— Verkettet zwei Listen
`(!!)`	::	$[\alpha] \to$ `Int` $\to \alpha$	— n-tes Element selektieren, gezählt ab 0
`concat`	::	$[[\alpha]] \to [\alpha]$	— "flachklopfen"
`length`	::	$[\alpha] \to$ `Int`	— Länge
`head, last`	::	$[\alpha] \to \alpha$	— Erstes bzw. letztes Element
`tail, init`	::	$[\alpha] \to [\alpha]$	— Hinterer bzw. vorderer Rest
`replicate`	::	`Int` $\to \alpha \to [\alpha]$	— Erzeuge n Kopien
`repeat`	::	$\alpha \to [\alpha]$	— Erzeugt zyklische Liste
`take, drop`	::	`Int` $\to [\alpha] \to [\alpha]$	— Erste bzw. letzte n Elemente
`splitAt`	::	`Int` $\to [\alpha] \to ([\alpha], [\alpha])$	— Spaltet an Index n, gezählt ab 0
`reverse`	::	$[\alpha] \to [\alpha]$	— Dreht Liste um
`zip`	::	$[\alpha] \to [\beta] \to [(\alpha, \beta)]$	— Erzeugt Liste von Paaren
`unzip`	::	$[(\alpha, \beta)] \to ([\alpha], [\beta])$	— Spaltet Liste von Paaren
`and, or`	::	`[Bool]` $\to$ `Bool`	— Konjunktion/Disjunktion
`sum, product`	::	`[Int]` $\to$ `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")
```

DEMO

Jetzt seit ihr dran!

Übung 4.2: Vordefinierte Typen

Sind folgende Ausdrücke typkorrekt, wenn ja welchen Typ haben sie, und was ist ihr Wert?

1. `take 4 (replicate 3 (3, 4))`
2. `snd (unzip (zip [1..10] "foo"))`
3. `"a"++ [('a')]`
4. `head [("foo", []), ("baz", 4 :: Integer)]`

Jetzt seit ihr dran!

Übung 4.2: Vordefinierte Typen

Sind folgende Ausdrücke typkorrekt, wenn ja welchen Typ haben sie, und was ist ihr Wert?

1. `take 4 (replicate 3 (3, 4))`
2. `snd (unzip (zip [1..10] "foo"))`
3. `"a"++ [('a')]`
4. `head [("foo", []), ("baz", 4 :: Integer)]`

Lösung:

1. Typ: `[(Integer, Integer)]`, Wert: `[(3,4),(3,4),(3,4)]`
2. Typ: `String`, Wert: `"foo"`
3. Typ: `String`, Wert: `"aa"`
4. Typfehler

III. Ad-Hoc Polymorphie

Parametrische Polymorphie: Grenzen

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

    Nicht auf allen Typen ist Gleichheit entscheidbar (besonders **Funktionen**)
  - Ordnung: `(≤)` $::\ \alpha \to\ \alpha \to$ `Bool`

    Nicht auf allen Typen definiert
  - Anzeige: `show` $::\ \alpha \to$ `String`

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

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)

Funktion `f` :: $\alpha \rightarrow \beta$ 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 $(\leq)$ (und andere Vergleiche)
- Anzeigen: `Show` für `show`
- Lesen: `Read` für `read` $::$ `String` $\rightarrow$ 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

Jetzt wieder ihr!

Übung 4.2: Meine Paare

Erinnert auch an die selbstgemachten Paare?

$\mathtt{data\ Pair}\ \alpha\ \beta = \mathtt{Pair\ \{\ left}\ ::\ \alpha,\ \mathtt{right}\ ::\ \beta\ \}$

Schreibt eine `Show`-Instanz, welches ein Tupel als `(a, b)` anzeigt!

Jetzt wieder ihr!

Übung 4.2: Meine Paare

Erinnert auch an die selbstgemachten Paare?

$\texttt{data Pair } \alpha\ \beta = \texttt{Pair \{ left :: } \alpha\texttt{, right :: } \beta\ \texttt{\}}$

Schreibt eine `Show`-Instanz, welches ein Tupel als `(a, b)` anzeigt!

Lösung:

- Voraussetzung: `Show a, Show b`
- Klammersetzung beachten

```
instance (Show a, Show b) ⇒ Show (Pair a b) where
  show (Pair a b) = "(" ++ show a ++ ", " ++ show b ++ ")"
```

IV. Typherleitung

Typen in Haskell (The Story So Far)

- Primitive Basisdatentypen: `Bool`, `Double`
- Funktionstypen: `Double` $\rightarrow$ `Int` $\rightarrow$ `Int`, `[Char]` $\rightarrow$ `Double`
- Typkonstruktoren: `[]`, `(...)`, `Foo`
- Typvariablen

$$
\begin{aligned}
\texttt{fst} &:: (\alpha, \beta) \rightarrow \alpha \\
\texttt{length} &:: [\alpha] \rightarrow \texttt{Int} \\
(+\!\!+) &:: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]
\end{aligned}
$$

- Typklassen :

$$
\begin{aligned}
\texttt{elem} &:: \texttt{Eq}\ \alpha \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \texttt{Bool} \\
\texttt{max} &:: \texttt{Ord}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha
\end{aligned}
$$

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
```

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

$$\begin{array}{cccccc} \texttt{f m xs} & = & \texttt{m} & \texttt{+} & \texttt{length} & \texttt{xs} \\ & & & & [\alpha] \to \texttt{Int} & \end{array}$$

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

$$
\begin{array}{cccccc}
\texttt{f m xs} & = & \texttt{m} & \texttt{+} & \texttt{length} & \texttt{xs} \\
 & & & & [\alpha] \to \texttt{Int} & \\
 & & & & & [\alpha]
\end{array}
$$

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

$$
\begin{array}{cccccc}
\texttt{f m xs} & = & \texttt{m} & \texttt{+} & \texttt{length} & \texttt{xs} \\
 & & & & [\alpha] \to \texttt{Int} & \\
 & & & & & [\alpha] \\
 & & & & \texttt{Int} &
\end{array}
$$

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`
				$[\alpha] \rightarrow$ `Int`	
					$[\alpha]$
				`Int`	
		`Int`			

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

$$
\begin{array}{cccccc}
\mathtt{f\ m\ xs} & = & \mathtt{m} & \mathtt{+} & \mathtt{length} & \mathtt{xs} \\
 & & & & [\alpha] \to \mathtt{Int} & \\
 & & & & & [\alpha] \\
 & & & & \mathtt{Int} & \\
 & & \mathtt{Int} & & & \\
 & & & \mathtt{Int} & & \\
\end{array}
$$

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

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
```

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		$[\beta] \to$ `Int`	γ

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**:

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

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**:

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

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**:

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

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		$[\beta] \to$ `Int`	γ	
				$[\beta]$	$\gamma \mapsto [\beta]$
			`Int`		
		`Int` $\to$ `Int` $\to$ `Int`			
	`Int`				$\alpha \mapsto$ `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**:

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

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		$[\beta] \to$ `Int`	γ	
				$[\beta]$	$\gamma \mapsto [\beta]$
			`Int`		
		`Int` $\to$ `Int` $\to$ `Int`			
	`Int`				$\alpha \mapsto$ `Int`
		`Int` $\to$ `Int`			
			`Int`		

`f` :: `Int` $\to [\beta] \to$ `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 $\mathit{DEXPTIME}(n)$ (deterministisch exponentiell), aber in der Praxis ist das **nie** ein Problem.

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

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

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

$$
\begin{array}{lccccc}
\texttt{f x y =} & \texttt{(x, 3)} & \texttt{:} & \texttt{('f', y)} & \texttt{:} & \texttt{[]} \\
 & \alpha \;\; \texttt{Int} & & \texttt{Char} \;\; \beta & & \texttt{[}\gamma\texttt{]}
\end{array}
$$

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

$$
\begin{array}{lccccc}
\texttt{f x y} = & \texttt{(x, 3)} & : & \texttt{('f', y)} & : & \texttt{[]} \\
 & \alpha\ \ \texttt{Int} & & \texttt{Char}\ \ \beta & & [\gamma] \\
 & (\alpha,\ \texttt{Int}) & & (\texttt{Char},\ \beta) & &
\end{array}
$$

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

$$
\begin{array}{llllll}
\texttt{f x y} = & \texttt{(x, 3)} & : & \texttt{('f', y)} & : & \texttt{[]} \\
 & \alpha \;\; \texttt{Int} & & \texttt{Char} \;\; \beta & & \texttt{[}\gamma\texttt{]} \\
 & \texttt{(}\alpha\texttt{, Int)} & & \texttt{(Char, }\beta\texttt{)} & & \\
 & & & \texttt{[(Char, }\beta\texttt{)]} & & \gamma \mapsto \texttt{(Char, }\beta\texttt{)}
\end{array}
$$

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

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

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

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

$$\texttt{f} :: \texttt{Char} \rightarrow \texttt{Int} \rightarrow [(\texttt{Char},\ \texttt{Int})]$$

- Allgemeinster Typ **muss nicht** existieren (Typfehler!)

Und was ist mit Typklassen?

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

$$\begin{array}{ll} \texttt{elem} & \texttt{3} \\ \texttt{Eq}\ \alpha\ ::\ \alpha \to [\alpha] \to \texttt{Bool} & \texttt{Num}\ \beta\ ::\ \beta \end{array}$$

$$\begin{array}{c} \texttt{elem 3} \\ (\texttt{Eq}\ \alpha,\ \texttt{Num}\ \alpha)\ ::\ [\alpha] \to \texttt{Bool} \end{array}$$

- Instantiierung muss Typklassen berücksichtigen:

$$\begin{array}{lll} \texttt{elem 3} & \texttt{"abc"} & \\ (\texttt{Eq}\ \alpha,\ \texttt{Num}\ \alpha)\ ::\ [\alpha] \to \texttt{Bool} & [\texttt{Char}] & \alpha \mapsto \texttt{Char} \end{array}$$

  - `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. Abschließende Bemerkungen

Polymorphie: the missing link

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

Polymorphie: the missing link

	Parametrisch	Ad-Hoc
Funktionen	`f :: α → Int`	`class F α where` `  f :: α → 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 $[\alpha]$
  - Optionen `Maybe` α
  - Tupel (α, β)