

# **Einführung in die Funktionale Programmierung:**

**Typisierung**

**Teil 2**

Prof Dr. Manfred Schmidt-Schauß

WS 2023/24

# Typisierung rekursiver Superkombinatoren

## Typisierung rekursiver Superkombinatoren

# Rekursive Superkombinatoren

## Beispiel

```
reverse xs                = reverseStack xs []  
reverseStack xs stack =  
    case xs of [] -> stack  
              (y:ys) -> reverseStack ys (y:stack)
```

# Rekursive Superkombinatoren

Definition (direkt rekursiv, rekursiv, verschränkt rekursiv)

- Sei  $\mathcal{SK}$  eine Menge von Superkombinatoren
- Für  $SK_i, SK_j \in \mathcal{SK}$  sei

$$SK_i \preceq SK_j$$

gdw.  $SK_j$  den Superkombinator  $SK_i$  im Rumpf benutzt.

- $\preceq^+$ : transitiver Abschluss von  $\preceq$  ( $\preceq^*$ : reflexiv-transitiver Abschluss)
- $SK_i$  ist **direkt rekursiv** wenn  $SK_i \preceq SK_i$  gilt.
- $SK_i$  ist **rekursiv** wenn  $SK_i \preceq^+ SK_i$  gilt.
- $SK_1, \dots, SK_m$  sind **verschränkt rekursiv**, wenn  $SK_i \preceq^+ SK_j$  für alle  $i, j \in \{1, \dots, m\}$

# Typisierung von nicht-rekursiven Superkombinatoren

- **Nicht**-rekursive Superkombinatoren kann man **wie Abstraktionen** typisieren
- Notation:  $A \vdash_T SK :: \tau$ , bedeutet:  
unter Annahme  $A$  kann man  $SK$  mit Typ  $\tau$  typisieren

# Typisierung von nicht-rekursiven Superkombinatoren

- **Nicht**-rekursive Superkombinatoren kann man **wie Abstraktionen** typisieren
- Notation:  $A \vdash_T SK :: \tau$ , bedeutet:  
unter Annahme  $A$  kann man  $SK$  mit Typ  $\tau$  typisieren

## Typisierungsregel für (geschlossene) nicht-rekursive SK:

$$(RSK1) \frac{A \cup \{x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \vdash_T SK :: \forall \mathcal{X}. \sigma(\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau)}$$

wenn  $\sigma$  Lösung von  $E$ ,

$SK \ x_1 \ \dots \ x_n = s$  die Definition von  $SK$

und  $SK$  nicht rekursiv ist,

und  $\mathcal{X}$  die Typvariablen in  $\sigma(\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau)$

$\tau$ -Notation:  $\tau$  steht für einen Typ innerhalb der Berechnung

## Beispiel: Typisierung von (.)

$$(\cdot) \quad f \circ g \circ x = f \circ (g \circ x)$$

$A_0$  ist leer, da keine Konstruktoren oder SK vorkommen.

$$\frac{\frac{\frac{(\text{AXV})\overline{A_1 \vdash g :: \alpha_2, \emptyset} \quad (\text{AXV})\overline{A_1 \vdash x :: \alpha_3, \emptyset}}{(\text{AXV})\overline{A_1 \vdash f :: \alpha_1, \emptyset} \quad (\text{RAPP})\overline{A_1 \vdash (g \ x) :: \alpha_5, \{\alpha_2 \dot{=} \alpha_3 \rightarrow \alpha_5\}}}{(\text{RAPP})\overline{A_1 \vdash (f \ (g \ x)) :: \alpha_4, \{\alpha_2 \dot{=} \alpha_3 \rightarrow \alpha_5, \alpha_1 = \alpha_5 \rightarrow \alpha_4\}}}{(\text{RSK1})\overline{\emptyset \vdash_T (. :: \forall \mathcal{X}. \sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4)}}$$

wobei  $A_1 = \{f :: \alpha_1, g :: \alpha_2, x :: \alpha_3\}$

Unifikation ergibt  $\sigma = \{\alpha_2 \mapsto (\alpha_3 \rightarrow \alpha_5), \alpha_1 \mapsto (\alpha_5 \rightarrow \alpha_4)\}$ .

Daher:  $\sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4) = (\alpha_5 \rightarrow \alpha_4) \rightarrow (\alpha_3 \rightarrow \alpha_5) \rightarrow \alpha_3 \rightarrow \alpha_4$

Jetzt kann man  $\mathcal{X} = \{\alpha_3, \alpha_4, \alpha_5\}$  berechnen, und umbenennen:

$$(\cdot) :: \forall a, b, c. (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$$

# Typisierung von rekursiven Superkombinatoren

- Sei  $SK\ x_1 \dots x_n = e$
- und  $SK$  kommt in  $e$  vor, d.h.  $SK$  ist rekursiv
- Warum kann man  $SK$  nicht ganz einfach typisieren?



# Typisierung von rekursiven Superkombinatoren

- Sei  $SK\ x_1 \ \dots \ x_n = e$
- und  $SK$  kommt in  $e$  vor, d.h.  $SK$  ist rekursiv
- Warum kann man  $SK$  nicht ganz einfach typisieren?
- Will man den Rumpf  $e$  typisieren, so muss man den Typ von  $SK$  schon kennen!

# Idee des Iterativen Typisierungsverfahrens

- Gebe  $SK$  zunächst den **allgemeinsten Typ** (d.h. eine Typvariable) und typisiere den Rumpf unter Benutzung dieses Typs
- Man erhält anschließend einen neuen Typ für  $SK$
- Mache mit neuem (quantifizierten) Typ im Rumpf weiter.
- Stoppe, wenn **neuer Typ = alter Typ**
- Dann hat man eine **konsistente Typannahme** gefunden; Diese ist dann der allgemeinste Typ.

**Allgemeinster Typ:** Typ  $T$  so dass  $\text{sem}(T) = \{\text{alle Grundtypen}\}$ .  
Das liefert der Typ  $\alpha$  (bzw. quantifiziert  $\forall \alpha. \alpha$ )

# Iteratives Typisierungsverfahren

## Regel zur Berechnung neuer Annahmen:

$$(\text{SKREK}) \frac{A \cup \{x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \vdash_T SK :: \sigma(\alpha_1 \rightarrow \dots \alpha_n \rightarrow \tau)}$$

wenn  $SK \ x_1 \ \dots \ x_n = s$  die Definition von  $SK$ ,  $\sigma$  Lösung von  $E$

Genau wie RSK1, aber in  $A$  muss es eine Annahme für  $SK$  geben.

# (Iteratives) Typisierungsverfahren: Vorarbeiten (1)

Die folgende Analyse der Aufrufhierarchie der Superkombinatoren wird im Hindley-Milner Typisierungsverfahren und in Haskell gebraucht!  
Im iterativen Typisierungsverfahren ist es eine Optimierung.

# Iteratives Typisierungsverfahren: Vorarbeiten (1)

Wegen verschränkter Rekursion:

- Abhängigkeitsanalyse der Superkombinatoren
- Berechnung der starken Zusammenhangskomponenten im Aufrufgraph
- Sei  $\simeq$  die Äquivalenzrelation passend zu  $\preceq^+$ , dann sind die starken Zusammenhangskomponenten gerade die Äquivalenzklassen zu  $\simeq$ .
- Jede Äquivalenzklasse wird gemeinsam typisiert

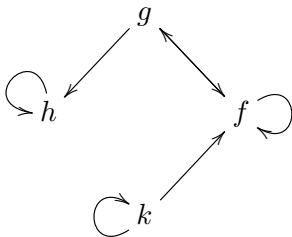
Typisierung der Gruppen entsprechend der  $\preceq^+$ -Ordnung modulo  $\simeq$ .

# Iteratives Typisierungsverfahren: Vorarbeiten (2)

Beispiel:

```
f x y = if x<=1 then y else f (x-y) (y + g x)
g x    = if x==0 then (f 1 x) + (h 2) else 10
h x    = if x==1 then 0 else h (x-1)
k x y = if x==1 then y else k (x-1) (y+(f x y))
```

Der Aufrufgraph (nur bzgl.  $f, g, h, k$ ) ist



Die Äquivalenzklassen (mit Ordnung) sind  $\{h\} \preceq^+ \{f, g\} \preceq^+ \{k\}$ .

# Iteratives Typisierungsverfahren: Der Algorithmus

## Iterativer Typisierungsalgorithmus

**Eingabe:** Menge von verschränkt rekursiven Superkombinatoren

$SK_1, \dots, SK_m$  wobei "kleinere" SK's schon typisiert; (keine freien Variablen)

- 1 Anfangsannahme  $A$  enthält Typen der Konstruktoren der bereits bekannten Superkombinatoren

# Iteratives Typisierungsverfahren: Der Algorithmus

## Iterativer Typisierungsalgorithmus

**Eingabe:** Menge von verschränkt rekursiven Superkombinatoren

$SK_1, \dots, SK_m$  wobei "kleinere" SK's schon typisiert; (keine freien Variablen)

- ① Anfangsannahme  $A$  enthält Typen der Konstruktoren der bereits bekannten Superkombinatoren
- ②  $A_0 := A \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$  und  $j = 0$ .



# Iteratives Typisierungsverfahren: Der Algorithmus

## Iterativer Typisierungsalgorithmus

**Eingabe:** Menge von verschränkt rekursiven Superkombinatoren

$SK_1, \dots, SK_m$  wobei "kleinere" SK's schon typisiert; (keine freien Variablen)

- ① Anfangsannahme  $A$  enthält Typen der Konstruktoren der bereits bekannten Superkombinatoren
- ②  $A_0 := A \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$  und  $j = 0$ .
- ③ Verwende für jeden Superkombinator  $SK_i$  (mit  $i = 1, \dots, m$ ) die Regel (SKREK) und Annahme  $A_j$ , um  $SK_i$  zu typisieren.

# Iteratives Typisierungsverfahren: Der Algorithmus

## Iterativer Typisierungsalgorithmus

**Eingabe:** Menge von verschränkt rekursiven Superkombinatoren

$SK_1, \dots, SK_m$  wobei "kleinere" SK's schon typisiert; (keine freien Variablen)

- ① Anfangsannahme  $A$  enthält Typen der Konstruktoren der bereits bekannten Superkombinatoren
- ②  $A_0 := A \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$  und  $j = 0$ .
- ③ Verwende für jeden Superkombinator  $SK_i$  (mit  $i = 1, \dots, m$ ) die Regel (SKREK) und Annahme  $A_j$ , um  $SK_i$  zu typisieren.
- ④ Wenn die  $m$  Typisierungen erfolgreich sind, d.h. für alle  $i$ :  
 $A_j \vdash_T SK_i :: \tau_i$   
Dann allquantifiziere:  $SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m$   
Setze  $A_{j+1} := A \cup \{SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m\}$

# Iteratives Typisierungsverfahren: Der Algorithmus

## Iterativer Typisierungsalgorithmus

**Eingabe:** Menge von verschränkt rekursiven Superkombinatoren

$SK_1, \dots, SK_m$  wobei "kleinere" SK's schon typisiert; (keine freien Variablen)

- ① Anfangsannahme  $A$  enthält Typen der Konstruktoren der bereits bekannten Superkombinatoren
- ②  $A_0 := A \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$  und  $j = 0$ .
- ③ Verwende für jeden Superkombinator  $SK_i$  (mit  $i = 1, \dots, m$ ) die Regel (SKREK) und Annahme  $A_j$ , um  $SK_i$  zu typisieren.
- ④ Wenn die  $m$  Typisierungen erfolgreich sind, d.h. für alle  $i$ :  
 $A_j \vdash_T SK_i :: \tau_i$   
 Dann allquantifiziere:  $SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m$   
 Setze  $A_{j+1} := A \cup \{SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m\}$
- ⑤ Wenn  $A_j \neq A_{j+1}$  (= Gleichheit bis auf Umbenennung), dann gehe mit  $j := j + 1$  zu Schritt (3). Anderenfalls, d.h. wenn  $A_j = A_{j+1}$ , war  $A_j$  konsistent; die Typen der  $SK_i$  sind entsprechend in  $A_j$  zu finden.

**Ausgabe** Die allquantifizierten polymorphen Typen der  $SK_i$

# Iteratives Typisierungsverfahren: Der Algorithmus

## Iterativer Typisierungsalgorithmus

**Eingabe:** Menge von verschränkt rekursiven Superkombinatoren

$SK_1, \dots, SK_m$  wobei "kleinere" SK's schon typisiert; (keine freien Variablen)

- ① Anfangsannahme  $A$  enthält Typen der Konstruktoren der bereits bekannten Superkombinatoren
- ②  $A_0 := A \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$  und  $j = 0$ .
- ③ Verwende für jeden Superkombinator  $SK_i$  (mit  $i = 1, \dots, m$ ) die Regel (SKREK) und Annahme  $A_j$ , um  $SK_i$  zu typisieren.
- ④ Wenn die  $m$  Typisierungen erfolgreich sind, d.h. für alle  $i$ :  
 $A_j \vdash_T SK_i :: \tau_i$   
 Dann allquantifiziere:  $SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m$   
 Setze  $A_{j+1} := A \cup \{SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m\}$
- ⑤ Wenn  $A_j \neq A_{j+1}$  (= Gleichheit bis auf Umbenennung), dann gehe mit  $j := j + 1$  zu Schritt (3). Anderenfalls, d.h. wenn  $A_j = A_{j+1}$ , war  $A_j$  konsistent; die Typen der  $SK_i$  sind entsprechend in  $A_j$  zu finden.

**Ausgabe** Die allquantifizierten polymorphen Typen der  $SK_i$

Wenn Fail auftritt, dann sind  $SK_1, \dots, SK_m$  nicht typisierbar.

# Eigenschaften des Algorithmus

- Die berechneten Typen pro Iterationsschritt sind **eindeutig bis auf Umbenennung**.  
⇒ bei Terminierung liefert der Algorithmus **eindeutige Typen**.
- Pro Iteration werden die neuen Typen **spezieller** (oder bleiben gleich).  
D.h. Monotonie bzgl. der Grundtypensemantik:  
 $\text{sem}(T_j) \supseteq \text{sem}(T_{j+1})$
- Bei Nichtterminierung gibt es **keinen polymorphen Typ**.  
Grund: Monotonie und man hat mit größten Annahmen begonnen.  
Und: “unendlicher” Schnitt ist leer.
- Das iterative Verfahren berechnet einen **größten Fixpunkt** (bzgl. der Grundtypensemantik): Menge wird solange verkleinert, bis sie sich nicht mehr ändert.  
D.h. es wird der **allgemeinste polymorphe Typ** berechnet

# Beispiele: length (1)

$$\text{length } xs = \text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow 0; (y : ys) \rightarrow 1 + \text{length } ys\}$$

Annahme:

$$A = \{\text{Nil} :: \forall a. [a], (:) :: \forall a. a \rightarrow [a] \rightarrow [a], 0, 1 :: \text{Int}, (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$$

$$1.\text{Iteration: } A_0 = A \cup \{\text{length} :: \forall \alpha. \alpha\}$$

$$\begin{array}{l}
 (a) \quad A_0 \cup \{xs :: \alpha_1\} \vdash xs :: \tau_1, E_1 \\
 (b) \quad A_0 \cup \{xs :: \alpha_1\} \vdash \text{Nil} :: \tau_2, E_2 \\
 (c) \quad A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (y : ys) :: \tau_3, E_3 \\
 (d) \quad A_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \tau_4, E_4 \\
 (e) \quad A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (1 + \text{length } ys) :: \tau_5, E_5 \\
 \hline
 (\text{RCASE}) \quad A_0 \cup \{xs :: \alpha_1\} \vdash (\text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow 0; (y : ys) \rightarrow 1 + \text{length } xs\}) :: \alpha_3, \\
 E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \dot{=} \tau_2, \tau_1 \dot{=} \tau_3, \alpha_3 \dot{=} \tau_4, \alpha_3 \dot{=} \tau_5\} \\
 \hline
 (\text{SKREK}) \quad A_0 \vdash_T \text{length} :: \sigma(\alpha_1 \rightarrow \alpha_3)
 \end{array}$$

wobei  $\sigma$  Lösung von

$$E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \dot{=} \tau_2, \tau_1 \dot{=} \tau_3, \alpha_3 \dot{=} \tau_4, \alpha_3 \dot{=} \tau_5\}$$

# Beispiele: length (2)

$$(a): \text{(AxV)} \frac{}{A_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset}$$

D.h.  $\tau_1 = \alpha_1$  und  $E_1 = \emptyset$

$$(b): \text{(AxK)} \frac{}{A_0 \cup \{xs :: \alpha_1\} \vdash \text{Nil} :: [\alpha_6], \emptyset}$$

D.h.  $\tau_2 = [\alpha_6]$  und  $E_2 = \emptyset$

$$(c) \frac{\text{(AxK)} \frac{}{A'_0 \vdash (: :: \alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9], \emptyset}, \text{(AxV)} \frac{}{A'_0 \vdash y :: \alpha_4, \emptyset}}{\text{(RAPP)} \frac{A'_0 \vdash ((: ) y) :: \alpha_8, \{\alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9] \dot{=} \alpha_4 \rightarrow \alpha_8\}}{\text{(RAPP)} \frac{A'_0 \vdash (y : ys) :: \alpha_7, \{\alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9] \dot{=} \alpha_4 \rightarrow \alpha_8, \alpha_8 \dot{=} \alpha_5 \rightarrow \alpha_7\}}}, \text{(AxV)} \frac{}{A'_0 \vdash ys :: \alpha_5, \emptyset}}$$

wobei  $A_0 = A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

D.h.  $\tau_3 = \alpha_7$  und  $E_3 = \{\alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9] \dot{=} \alpha_4 \rightarrow \alpha_8, \alpha_8 \dot{=} \alpha_5 \rightarrow \alpha_7\}$

# Beispiele: length (3)

(d) 
$$\text{(AxK)} \frac{}{A_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \text{Int}, \emptyset}$$
  
 D.h.  $\tau_4 = \text{Int}$  und  $E_4 = \emptyset$

(e) 
$$\text{(RApP)} \frac{\text{(AxK)} \frac{}{A'_0 \vdash (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \emptyset}, \text{(AxK)} \frac{}{A'_0 \vdash 1 :: \text{Int}, \emptyset}}{A'_0 \vdash ((+) 1) :: \alpha_{11}, \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \dot{=} \text{Int} \rightarrow \alpha_{11}\}}, \text{(RApP)} \frac{\text{(AxSK)} \frac{}{A'_0 \vdash \text{length} :: \alpha_{13}, \emptyset}, \text{(AxV)} \frac{}{A'_0 \vdash (ys) :: \alpha_5, \emptyset}}{A'_0 \vdash (\text{length } ys) :: \alpha_{12}, \{\alpha_{13} \dot{=} \alpha_5 \rightarrow \alpha_{12}\}}}{A'_0 \vdash (1 + \text{length } ys) :: \alpha_{10}, \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \dot{=} \text{Int} \rightarrow \alpha_{11}, \alpha_{13} \dot{=} \alpha_5 \rightarrow \alpha_{12}, \alpha_{11} \dot{=} \alpha_{12} \rightarrow \alpha_{10}\}}$$
  
 wobei  $A_0 = A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

D.h.  $\tau_5 = \alpha_{10}$  und

$$E_5 = \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \dot{=} \text{Int} \rightarrow \alpha_{11}, \alpha_{13} \dot{=} \alpha_5 \rightarrow \alpha_{12}, \alpha_{11} \dot{=} \alpha_{12} \rightarrow \alpha_{10}\}$$



# Beispiele: length (3)

Zusammengefasst:

$$A_0 \vdash_T \text{length} :: \sigma(\alpha_1 \rightarrow \alpha_3)$$

wobei  $\sigma$  Lösung von

$$\begin{aligned} &\{\alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9] \doteq \alpha_4 \rightarrow \alpha_8, \alpha_8 \doteq \alpha_5 \rightarrow \alpha_7, \\ &\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \doteq \text{Int} \rightarrow \alpha_{11}, \alpha_{13} \doteq \alpha_5 \rightarrow \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \rightarrow \alpha_{10}, \\ &\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \text{Int}, \alpha_3 \doteq \alpha_{10}\} \end{aligned}$$

Die Unifikation ergibt als Unifikator

$$\{\alpha_1 \mapsto [\alpha_9], \alpha_3 \mapsto \text{Int}, \alpha_4 \mapsto \alpha_9, \alpha_5 \mapsto [\alpha_9], \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto [\alpha_9], \alpha_8 \mapsto [\alpha_9] \rightarrow [\alpha_9], \\ \alpha_{10} \mapsto \text{Int}, \alpha_{11} \mapsto \text{Int} \rightarrow \text{Int}, \alpha_{12} \mapsto \text{Int}, \alpha_{13} \mapsto [\alpha_9] \rightarrow \text{Int}\}$$

$$\text{daher } \sigma(\alpha_1 \rightarrow \alpha_3) = [\alpha_9] \rightarrow \text{Int}$$

$$A_1 = A \cup \{\text{length} :: \forall \alpha. [\alpha] \rightarrow \text{Int}\}$$

Da  $A_0 \neq A_1$  muss man mit  $A_1$  erneut iterieren.

2.Iteration: Ergibt den gleichen Typ, daher war  $A_1$  konsistent.

# Iterative Typisierung von `length`, vereinfacht

$$\text{length } xs = \text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow 0; (y : ys) \rightarrow 1 + \text{length } ys\}$$

1. Iteration:  $A_0 = A \cup \{\text{length} :: \forall \alpha. \alpha\}$

bekannt:  $\{\text{Nil} :: \forall a. [a], (:) :: \forall a. a \rightarrow [a] \rightarrow [a], 0, 1 :: \text{Int}, (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$

Bedingungen ergeben:

- $xs :: [\alpha]$
- $\text{length } ys$  hat keine Beschränkungen;  
Also:  $(\text{case } \dots) :: \text{Int}$ .
- Erste Iteration:  $\text{length} :: [\alpha] \rightarrow \text{Int}$ .

# Iterative Typisierung von length, vereinfacht

**Zweiter Schritt:** mit  $\text{length} :: (\forall \alpha : [\alpha] \rightarrow \text{Int})$  im Rumpf

$\text{length } xs = \text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow 0; (y : ys) \rightarrow 1 + \text{length } ys\}$

2.Iteration:  $A_1 = A \cup \{\text{length} :: \forall \alpha. [\alpha] \rightarrow \text{Int}\}$

bekannt:  $\{\text{Nil} :: \forall a. [a], (:) :: \forall a. a \rightarrow [a] \rightarrow [a], 0, 1 :: \text{Int}, (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$

Bedingungen ergeben:

- $xs :: [\alpha]$
- $\text{length } ys$  erzwingt:  $ys :: [\alpha]$ ; und dann  $y :: \alpha$ .  
Alle Typen passen!  
Also:  $(\text{case } \dots) :: \text{Int}$ .
- Zweite Iteration ergibt ebenfalls:  $\text{length} :: [\alpha] \rightarrow \text{Int}$ .  
Und das war die Annahme. Also Fixpunkt gefunden.

# Iteratives Verfahren ist allgemeiner als Haskell

## Beispiel

$g \ x = 1 : (g \ (g \ 'c'))$

$A = \{1 :: \text{Int}, \text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], 'c' :: \text{Char}\}$

$A_0 = A \cup \{g :: \forall \alpha. \alpha\}$  (und  $A'_0 = A_0 \cup \{x :: \alpha_1\}$ ):

$$\begin{array}{c}
 \text{(AxSK)} \frac{}{A'_0 \vdash g :: \alpha_8, \emptyset}, \quad \text{(AxK)} \frac{}{A'_0 \vdash 'c' :: \text{Char}}, \\
 \text{(RAPP)} \frac{}{A'_0 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, \quad \text{(AxSK)} \frac{}{A'_0 \vdash 1 :: \text{Int}, \emptyset}, \quad \text{(RAPP)} \frac{}{A'_0 \vdash g :: \alpha_6, \emptyset}, \quad \text{(RAPP)} \frac{}{A'_0 \vdash (g \ 'c') :: \alpha_7, \{\alpha_8 \doteq \text{Char} \rightarrow \alpha_7\}}, \\
 \text{(RAPP)} \frac{}{A'_0 \vdash (\text{Cons } 1) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3}, \quad \text{(RAPP)} \frac{}{A'_0 \vdash (g \ (g \ 'c')) :: \alpha_4, \{\alpha_8 \doteq \text{Char} \rightarrow \alpha_7, \alpha_6 \doteq \alpha_7 \rightarrow \alpha_4\}}, \\
 \text{(SKREK)} \frac{}{A'_0 \vdash \text{Cons } 1 \ (g \ (g \ 'c')) :: \alpha_2, \{\alpha_8 \doteq \text{Char} \rightarrow \alpha_7, \alpha_6 \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}}, \\
 A_0 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_1 \rightarrow [\text{Int}]
 \end{array}$$

wobei  $\sigma = \{\alpha_2 \mapsto [\text{Int}], \alpha_3 \mapsto [\text{Int}] \rightarrow [\text{Int}], \alpha_4 \mapsto [\text{Int}], \alpha_5 \mapsto \text{Int}, \alpha_6 \mapsto \alpha_7 \rightarrow [\text{Int}], \alpha_8 \mapsto \text{Char} \rightarrow \alpha_7\}$  die Lösung v  
 $\{\alpha_8 \doteq \text{Char} \rightarrow \alpha_7, \alpha_6 \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}$  ist.

D.h.  $A_1 = A \cup \{g :: \forall \alpha. \alpha \rightarrow [\text{Int}]\}$ .

Nächste Iteration zeigt:  $A_1$  ist konsistent.

# Iteratives Verfahren ist allgemeiner als Haskell (2)

Beachte: Für die Funktion `g` kann Haskell keinen Typ herleiten:

```
Prelude> let g x = 1:(g (g 'c'))
```

```
<interactive>:1:13:
```

```
Couldn't match expected type '[t]' against inferred type 'Char'
```

```
Expected type: Char -> [t]
```

```
Inferred type: Char -> Char
```

```
In the second argument of '(:)', namely '(g (g 'c'))'
```

```
In the expression: 1 : (g (g 'c'))
```

# Iteratives Verfahren ist allgemeiner als Haskell (2)

Beachte: Für die Funktion  $g$  kann Haskell keinen Typ herleiten:

```
Prelude> let g x = 1:(g (g 'c'))
```

```
<interactive>:1:13:
```

```
Couldn't match expected type '[t]' against inferred type 'Char'
```

```
Expected type: Char -> [t]
```

```
Inferred type: Char -> Char
```

```
In the second argument of '(:)', namely '(g (g 'c'))'
```

```
In the expression: 1 : (g (g 'c'))
```

Aber: Haskell kann den Typ verifizieren, wenn man ihn angibt:

```
let g::a -> [Int]; g x = 1:(g (g 'c'))
```

```
Prelude> :t g
```

```
g :: a -> [Int]
```

Grund: Wenn Typ vorhanden, führt Haskell keine Typinferenz durch, sondern verifiziert nur die Annahme.

$g$  wird im Rumpf wie bereits typisiert behandelt.

Bsp.: Mehrere Iterationen sind nötig (1)

$$gg\ x = x : (g\ (g\ 'c'))$$

- $A = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], 'c' :: \text{Char}\}.$
- $A_0 = A \cup \{g :: \forall \alpha. \alpha\}$

$$\frac{\frac{\frac{(\text{AxK})}{A'_0 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, \frac{(\text{AxV})}{A'_0 \vdash x :: \alpha_1, \emptyset}}{(\text{RApp})} \quad \frac{\frac{(\text{AxSK})}{A'_0 \vdash g :: \alpha_6, \emptyset}, \frac{(\text{RApp})}{A'_0 \vdash (g \text{ 'c'}) :: \alpha_7, \{\alpha_8 \doteq \text{Char} \rightarrow \alpha_7\}}}{(\text{RApp})} \quad \frac{(\text{SKREK})}{A'_0 \vdash \text{Cons } x (g (g \text{ 'c'})) :: \alpha_2, \{\alpha_8 \doteq \text{Char} \rightarrow \alpha_7, \alpha_6 \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}}$$

$A_0 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_5 \rightarrow [\alpha_5]$   
 wobei  $\sigma = \{\alpha_1 \mapsto \alpha_5, \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [\alpha_5] \rightarrow [\alpha_5], \alpha_4 \mapsto [\alpha_5], \alpha_6 \mapsto \alpha_7 \rightarrow [\alpha_5], \alpha_8 \mapsto \text{Char} \rightarrow \alpha_7\}$  die Lösung von  
 $\{\alpha_8 \doteq \text{Char} \rightarrow \alpha_7, \alpha_6 \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}$  ist.

$$\text{D.h. } A_1 = A \cup \{g :: \forall \alpha. \alpha \rightarrow [\alpha]\}.$$

Bsp.: Mehrere Iterationen sind nötig (2)

Da  $A_0 \neq A_1$  muss eine weitere Iteration durchgeführt werden.

mit  $g :: \forall \alpha. \alpha \rightarrow [\alpha]$

$$g \cdot x = x : (g \cdot (g \cdot c'))$$

Sei  $A'_1 = A_1 \cup \{x :: \alpha_1\}$ :

$$\frac{\begin{array}{c} \text{(AXK)} \\ A'_1 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset, \quad A'_1 \vdash x :: \alpha_1, \emptyset \\ \text{(RAPP)} \\ A'_1 \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3 \end{array}, \quad \frac{\begin{array}{c} \text{(AXSK)} \\ A'_1 \vdash g :: \alpha_6 \rightarrow [\alpha_6], \emptyset, \quad A'_1 \vdash (g \text{ 'c' }) :: \alpha_7, \{\alpha_8 \rightarrow [\alpha_8]\dot{=} \text{Char} \rightarrow \alpha_7\} \\ \text{(RAPP)} \\ A'_1 \vdash (g (g \text{ 'c' })) :: \alpha_4, \{\alpha_8 \rightarrow [\alpha_8]\dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6]\dot{=} \alpha_7 \rightarrow \alpha_4\} \end{array}}{\begin{array}{c} \text{(RAPP)} \\ A'_1 \vdash \text{Cons } x (g (g \text{ 'c' })) :: \alpha_2, \{\alpha_8 \rightarrow [\alpha_8]\dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6]\dot{=} \alpha_7 \rightarrow \alpha_4 \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5]\dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\} \\ \text{(SKREK)} \\ A_1 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = [\text{Char}] \rightarrow [[\text{Char}]] \end{array}}$$

wobei  $\sigma = \{\alpha_1 \mapsto [\text{Char}], \alpha_2 \mapsto [[\text{Char}]], \alpha_3 \mapsto [[\text{Char}]] \rightarrow [[\text{Char}]], \alpha_4 \mapsto [[[\text{Char}]]], \alpha_5 \mapsto [\text{Char}], \alpha_6 \mapsto [\text{Char}], \alpha_7 \mapsto [\text{Char}], \alpha_8 \mapsto \text{Char}]$   
die Lösung von  $\{\alpha_8 \rightarrow [\alpha_8] \doteq \text{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6] \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}$  ist.

Daher ist  $A_2 = A \cup \{g :: [\text{Char}] \rightarrow [[\text{Char}]]\}$ .



# Bsp.: Mehrere Iterationen sind nötig (3)

$g \ x = x : (g \ (g \ 'c'))$

Da  $A_1 \neq A_2$  muss eine weitere Iteration durchgeführt werden:

Sei  $A'_2 = A_2 \cup \{x :: \alpha_1\} = A \cup \{g :: [\text{Char}] \rightarrow [[\text{Char}]], x :: \alpha_1\}$

$$\begin{array}{c}
 \frac{(A_{\text{SK}}) \overline{A'_2 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, (A_{\text{SV}}) \overline{A'_2 \vdash x :: \alpha_1, \emptyset}}{(RA_{\text{APP}})} \overline{A'_2 \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3} \quad , \quad \frac{(A_{\text{SK}}) \overline{A'_2 \vdash g :: [\text{Char}] \rightarrow [[\text{Char}]], \emptyset}, (RA_{\text{APP}}) \overline{A'_2 \vdash (g \ 'c') :: \alpha_7, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7\}}}{(RA_{\text{APP}})} \overline{A'_2 \vdash (g \ (g \ 'c')) :: \alpha_4, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7, [\text{Char}] \rightarrow [[\text{Char}]] \doteq \alpha_7 \rightarrow \alpha_4\}} \\
 \frac{(SK_{\text{REC}}) \overline{A'_2 \vdash \text{Cons } x \ (g \ (g \ 'c')) :: \alpha_2, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7, [\text{Char}] \rightarrow [[\text{Char}]] \doteq \alpha_7 \rightarrow \alpha_4 \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}}{A_2 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2)} \\
 \text{wobei } \sigma \text{ die Lösung von} \\
 \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7, [\text{Char}] \rightarrow [[\text{Char}]] \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\} \text{ ist.}
 \end{array}$$

Unifikation:

$$\begin{array}{c}
 [\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7, \\
 \dots \\
 \hline
 [\text{Char}] \doteq \text{Char}, \\
 [[\text{Char}]] \doteq \alpha_7, \\
 \dots \\
 \hline
 \text{Fail}
 \end{array}$$

$g$  ist nicht typisierbar.

# Daher gilt ...

## Beobachtung

Das iterative Typisierungsverfahren benötigt unter Umständen mehrere Iterationen, bis ein Ergebnis (untypisiert / konsistente Annahme) gefunden wurde.

Beachte: Es gibt auch Beispiele, die zeigen, dass mehrere Iterationen nötig sind, um eine konsistente Annahme zu finden (Übungsaufgabe).

# Nichtterminierung des iterativen Verfahrens

$$f = [g]$$

$$g = [f]$$

Es gilt  $f \simeq g$ , d.h. das iterative Verfahren typisiert  $f$  und  $g$  gemeinsam.

$$A = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} : \forall a. a\}.$$

$$A_0 = A \cup \{f :: \forall \alpha. \alpha, g :: \forall \alpha. \alpha\}$$

$$\begin{array}{c}
 (\text{AxK}) \frac{}{A_0 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad (\text{AxSK}) \frac{}{A_0 \vdash g :: \alpha_5} \\
 (\text{RAPP}) \frac{}{A_0 \vdash (\text{Cons } g) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3\}}, \quad (\text{AxK}) \frac{}{A_0 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 (\text{RAPP}) \frac{}{A_0 \vdash [g] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}} \\
 (\text{SKREK}) \frac{}{A_0 \vdash_T f :: \sigma(\alpha_1) = [\alpha_5]}
 \end{array}$$

$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \rightarrow [\alpha_5], \alpha_4 \mapsto \alpha_5\}$  ist  
 Lösung von  $\{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}$

# Nichtterminierung des iterativen Verfahrens (2)

$$\begin{array}{c}
 \text{(AXK)} \frac{}{A_0 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad \text{(AXSK)} \frac{}{A_0 \vdash \mathbf{f} :: \alpha_5} \\
 \text{(RAPP)} \frac{}{A_0 \vdash (\text{Cons } \mathbf{f}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} \alpha_5 \rightarrow \alpha_3\}}, \quad \text{(AXK)} \frac{}{A_0 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{A_0 \vdash [\mathbf{f}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} \alpha_5 \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SKREK)} \frac{}{A_0 \vdash_T \mathbf{g} :: \sigma(\alpha_1) = [\alpha_5]}
 \end{array}$$

$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \rightarrow [\alpha_5], \alpha_4 \mapsto \alpha_5\}$  ist  
 Lösung von  $\{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} \alpha_5 \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}$

Daher ist  $A_1 = A \cup \{\mathbf{f} :: \forall a. [a], \mathbf{g} :: \forall a. [a]\}$ . Da  $A_1 \neq A_0$  muss man weiter iterieren.

# Nichtterminierung des iterativen Verfahrens (3)

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_1 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad \text{(AxSK)} \frac{}{A_1 \vdash g :: [\alpha_5]} \\
 \text{(RAPP)} \frac{}{A_1 \vdash (\text{Cons } g) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3\}}, \quad \text{(AxK)} \frac{}{A_1 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{A_1 \vdash [g] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SKREK)} \frac{}{A_1 \vdash_T f :: \sigma(\alpha_1) = [[\alpha_5]]}
 \end{array}$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \rightarrow [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$  ist

Lösung von  $\{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}$

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_1 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad \text{(AxSK)} \frac{}{A_1 \vdash f :: [\alpha_5]} \\
 \text{(RAPP)} \frac{}{A_1 \vdash (\text{Cons } f) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3\}}, \quad \text{(AxK)} \frac{}{A_1 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{A_1 \vdash [f] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SKREK)} \frac{}{A_1 \vdash_T g :: \sigma(\alpha_1) = [[\alpha_5]]}
 \end{array}$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \rightarrow [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$  ist

Lösung von  $\{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}$

Daher ist  $A_2 = A \cup \{f :: \forall a. [[a]], g :: \forall a. [[a]]\}$ . Da  $A_2 \neq A_1$  muss man weiter iterieren.

# Nichtterminierung des iterativen Verfahrens (4)

Vermutung: Terminiert nicht

Beweis: (Induktion) betrachte den  $i$ . Schritt:

$A_i = A \cup \{f :: \forall a. [a]^i, g :: \forall a. [a]^i\}$  wobei  $[a]^i$   $i$ -fach geschachtelte Liste

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_i \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad \text{(AxSK)} \frac{}{A_i \vdash g :: [\alpha_5]^i} \\
 \text{(RAPP)} \frac{}{A_i \vdash (\text{Cons } g) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3\}}, \quad \text{(AxK)} \frac{}{A_i \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{A_i \vdash [g] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SKREK)} \frac{}{}
 \end{array}$$

$$A_i \vdash_T f :: \sigma(\alpha_1) = [[\alpha_5]^i]$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \rightarrow [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$  ist

Lösung von  $\{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}$

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_i \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad \text{(AxSK)} \frac{}{A_i \vdash f :: [\alpha_5]^i} \\
 \text{(RAPP)} \frac{}{A_i \vdash (\text{Cons } f) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3\}}, \quad \text{(AxK)} \frac{}{A_i \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{A_i \vdash [f] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SKREK)} \frac{}{}
 \end{array}$$

$$A_i \vdash_T g :: \sigma(\alpha_1) = [[\alpha_5]^i]$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \rightarrow [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$  ist

Lösung von  $\{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}$

D.h.  $A_{i+1} = A \cup \{f :: \forall a. [a]^{i+1}, g :: \forall a. [a]^{i+1}\}.$

# Daher ...

## Beobachtung

Das iterative Typisierungsverfahren terminiert nicht immer.

# Daher ...

## Beobachtung

Das iterative Typisierungsverfahren terminiert nicht immer.

Es gilt sogar:

## Satz

*Die iterative Typisierung ist unentscheidbar.*

Dies folgt aus der Unentscheidbarkeit der so genannten Semi-Unifikation von First-Order Termen.  
(siehe Forschungsliteratur)



# Aufrufhierarchie

- Das iterative Verfahren benötigt die Information aus der Aufrufhierarchie nicht:
- Es liefert die gleichen Typen, unabhängig davon, in welcher Reihenfolge man die SK typisiert.
- Die Aufrufhierarchie ist aber notwendig für das Hindley-Milner Verfahren (kommt noch).

# Type Safety (einer Programmiersprache)

Man spricht von **Type Safety** wenn gilt:

- („Type Preservation“)

Die Typisierung bleibt unter Reduktion erhalten

Für einen Grundtyp  $\tau$ :

Wenn  $t :: \tau$  vor der Reduktion  $t \rightarrow t'$ ,

dann auch  $t' :: \tau$  danach.

D.h. Typen der Ausdrücke können allgemeiner werden.

- („Progress Lemma“):

Getypte geschlossene Ausdrücke sind reduzibel, solange sie keine WHNF sind .

# Type Safety (2)

## Lemma

*Sei  $s$  ein direkt dynamisch ungetypter KFPTS+seq-Ausdruck.  
Dann kann das iterative Typsystem keinen Typ für  $s$  herleiten.*

**Beweis:**  $s$  direkt dynamisch ungetypt ist, gdw.:

- $s = R[\text{case}_T (c \ s_1 \ \dots \ s_n) \text{ of } \text{Alts}]$  und  $c$  ist nicht vom Typ  $T$ .  
Typisierung von  $\text{case}$  fügt Gleichungen hinzu, so dass der Typ von  $(c \ s_1 \ \dots \ s_n)$  und Typ von Pattern gleich ist. Daher wird die Unifikation scheitern.
- $s = R[\text{case}_T \ \lambda x.t \text{ of } \text{Alts}]$ : Analog, Gleichungen verlangen dass  $(\lambda x.t)$  einen Funktionstyp erhält, Pattern aber nie einen solchen haben.
- $R[(c \ s_1 \ \dots \ s_{\text{ar}(c)}) \ t]$ : Typisierung typisiert die Anwendung  $((c \ s_1 \ \dots \ s_{\text{ar}(c)}) \ t)$  wie eine verschachtelte Anwendung  $((((c \ s_1) \ \dots) \ s_{\text{ar}(c)}) \ t)$ . Es werden Gleichungen hinzugefügt, die sicherstellen, dass  $c$  höchstens  $\text{ar}(c)$  Argumente verarbeiten kann.

# Type Safety (3)

## Lemma (Type Preservation)

*Sei  $s$  ein wohl-getypter, geschlossener KFPTS+seq-Ausdruck und  $s \xrightarrow{no} s'$ . Dann ist  $s'$  wohl-getypt.*

**Beweis:** Hierfür muss man die einzelnen Fälle einer  $(\beta)$ -,  $(SK-\beta)$ - und  $(\text{case})$ -Reduktion durchgehen. Für die Typherleitung von  $s$  kann man aus der Typherleitung einen Typ für jeden Unterterm von  $s$  ablesen. Bei der Reduktion werden diese Typen einfach mitkopiert.

# Type Safety (4)

Aus den letzten beiden Lemmas folgt:

## Satz

Sei  $s$  ein wohl-getypter, geschlossener KFPTS+seq-Ausdruck.  
Dann ist  $s$  nicht dynamisch ungetypt.

## Lemma (Progress Lemma)

Sei  $s$  ein wohl-getypter, geschlossener KFPTS+seq-Ausdruck.  
Dann gilt:

- $s$  ist eine WHNF, oder
- $s$  ist normalordnungsreduzibel, d.h.  $s \xrightarrow{no} s'$ .

**Beweis** Betrachtet man die Fälle, wann ein geschlossener KFPTS+seq-Ausdruck irreduzibel ist, so erhält man:  
 $s$  ist eine WHNF oder  $s$  ist direkt-dynamisch ungetypt. Daher folgt das Lemma.

# Type Safety (5)

## Satz

*Die iterative Typisierung für  $KFPTS+seq$  erfüllt die „Type-safety“-Eigenschaft.*

## Hindley-Milner Typisierung

Roger Hindley; Robin Milner und Luis Damas haben beigetragen.

# Erzwingen der Terminierung (der Typcheck-Iteration)

- $SK_1, \dots, SK_m$  ist Gruppe verschränkt rekursiver Superkombinatoren
- $A_i \vdash_T SK_1 :: \tau_1, \dots, A_i \vdash_T SK_m :: \tau_m$  seien die durch die  $i$ . Iteration hergeleiteten Typen

**Hindley-Milner-Schritt:** Typisiere  $SK_1, \dots, SK_m$  gemeinsam in einem Schritt,  
mit der Annahme:

$$A_M = A \cup \{SK_1 :: \tau_1, \dots, SK_m :: \tau_m\};$$

**ohne Quantoren**

D.h.: keine umbenannten Kopien der Typen bei verschiedenen Vorkommen des gleichen Namens



# Erzwingen der Terminierung (2)

für  $i = 1, \dots, m$ :

$$\begin{array}{l}
 \text{(SKREKM)} \quad \frac{A_M \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau'_i, E_i}{A_M \vdash_T \text{ für } i = 1, \dots, m \quad SK_i :: \sigma(\alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau'_i)} \\
 \text{wenn } \sigma \text{ Lösung von } E_1 \cup \dots \cup E_m \cup \bigcup_{i=1}^m \{\tau_i \doteq \alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau'_i\} \\
 \text{und } SK_1 \ x_{1,1} \ \dots \ x_{1,n_1} \quad = \quad s_1 \\
 \quad \quad \quad \dots \\
 \quad \quad \quad SK_m \ x_{m,1} \ \dots \ x_{m,n_m} \quad = \quad s_m \\
 \text{die Definitionen von } SK_1, \dots, SK_m \text{ sind}
 \end{array}$$

Als zusätzliche Regel muss im Typisierungsverfahren hinzugefügt werden:

$$\begin{array}{l}
 \text{(AxSK2)} \quad \frac{}{A \cup \{SK :: \tau\} \vdash SK :: \tau} \\
 \text{wenn } \tau \text{ nicht allquantifiziert ist}
 \end{array}$$

# Erzwingen der Terminierung (3)

Unterschied zum iterativen Schritt:

- Die Typen der zu typisierenden SKs werden nicht allquantifiziert.  
(allquantifiziert sind die bekannten Typen von anderen SKs. )
- Daher sind während der Typisierung **keine Kopien** der aktuell zu bestimmenden Typen möglich
- Am Ende werden die **angenommenen** Typen mit den **hergeleiteten** Typen unifiziert.

# Erzwingen der Terminierung (3)

Unterschied zum iterativen Schritt:

- Die Typen der zu typisierenden SKs werden nicht allquantifiziert.  
(allquantifiziert sind die bekannten Typen von anderen SKs. )
- Daher sind während der Typisierung **keine Kopien** der aktuell zu bestimmenden Typen möglich
- Am Ende werden die **angenommenen** Typen mit den **hergeleiteten** Typen unifiziert.

Daraus folgt:

Die neue Annahme, die man durch die (SKREKM)-Regel herleiten kann, ist **stets konsistent**.

Nach einem Hindley-Milner-Schritt terminiert das Verfahren sofort.

# Das Hindley-Milner-Typisierungsverfahren

Hindley-Milner-Typisierung ist „Spezialisierung“ des iterativen Typisierungsverfahren.

## Unterschiede:

- Es wird nur ein Iterationsschritt durchgeführt.
- Die aktuell zu typisierenden Superkombinatoren  $SK_i$  sind mit allgemeinstem Typ  $\alpha_i$  (ohne Allquantor) in den Annahmen.
- Hindley-Milner typisierbar impliziert iterativ typisierbar

## Effekte:

- Hindley-Milner ist schneller;
- es kann weniger Ausdrücke typisieren;
- und liefert manchmal speziellere Typen

# Das Hindley-Milner-Typisierungsverfahren

Hindley-Milner-Typisierung ist „Spezialisierung“ des iterativen Typisierungsverfahren.

## Unterschiede:

- Es wird nur ein Iterationsschritt durchgeführt.
- Die aktuell zu typisierenden Superkombinatoren  $SK_i$  sind mit allgemeinstem Typ  $\alpha_i$  (ohne Allquantor) in den Annahmen.
- Hindley-Milner typisierbar impliziert iterativ typisierbar

## Effekte:

- Hindley-Milner ist schneller;
- es kann weniger Ausdrücke typisieren;
- und liefert manchmal speziellere Typen

Haskell verwendet das Hindley-Milner-Typisierungs-Verfahren.  
Allerdings erweitert...

# Das Hindley-Milner-Typisierungsverfahren, genauer

## Hindley-Milner-Typisierungsverfahren:

$SK_1, \dots, SK_m$  ist eine  $\simeq$ -Äquivalenzklasse wobei alle echt  $\simeq$  kleineren SKs bereits getypt sind.

- ① Annahme  $A$  enthält Typen der bereits typisierten SKs und Konstruktoren (allquantifiziert)
- ② Typisiere  $SK_1, \dots, SK_m$  mit der Regel (MSKREK):

für  $i = 1, \dots, m$ :

$$(MSKREK) \frac{A \cup \{SK_1 :: \beta_1, \dots, SK_m :: \beta_m\} \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i, E_i}{A \vdash_T \text{ für } i = 1, \dots, m \quad SK_i :: \sigma(\alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau_i)}$$

wenn  $\sigma$  Lösung von  $E_1 \cup \dots \cup E_m \cup \bigcup_{i=1}^m \{\beta_i \doteq \alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau_i\}$

$$\text{und } SK_1 \ x_{1,1} \ \dots \ x_{1,n_1} = s_1$$

...

$$SK_m \ x_{m,1} \ \dots \ x_{m,n_m} = s_m$$

die Definitionen von  $SK_1, \dots, SK_m$  sind

Falls Unifikation fehlschlägt, sind  $SK_1, \dots, SK_m$  nicht Hindley-Milner-typisierbar

# Das Hindley-Milner-Typisierungsverfahren (2)

Vereinfachung: Regel für einen (unabhängigen) rekursiven SK

$$\begin{array}{c}
 \text{(MSKREK1)} \quad \frac{A \cup \{SK :: \beta, x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \vdash_T SK :: \sigma(\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau)} \\
 \text{wenn } \sigma \text{ Lösung von } E \cup \{\beta \doteq \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau\} \\
 \text{und } SK \ x_1 \ \dots \ x_n = s \text{ die Definition von } SK \text{ ist}
 \end{array}$$

# Eigenschaften des Hindley-Milner-Typcheck

Für das Hindley-Milner-Typisierungsverfahren gelten die folgenden Eigenschaften:

- Das Verfahren **terminiert**.
- Das Verfahren liefert eindeutige Typen (bis auf Umbenennung von Variablen)
- Die Hindley-Milner-Typisierung ist **entscheidbar**.
- Das Problem, ob ein Ausdruck Hindley-Milner-typisierbar ist, ist **DEXPTIME-vollständig**
- Das Verfahren liefert u.U. eingeschränktere Typen als das iterative Verfahren. Insbesondere kann ein Ausdruck iterativ typisierbar, aber nicht Hindley-Milner-typisierbar sein.
- Das Hindley-Milner-Typisierungsverfahren benötigt das Wissen um die **Aufrufhierarchie der Superkombinatoren**: Es berechnet evtl. weniger allgemeine Typen bzw. Typisierung schlägt fehl, wenn man nicht von unten nach oben typisiert.



# Beispiele: Viele Typvariablen

Man benötigt manchmal exponentiell viele Typvariablen  
(in der Größe des Ausdrucks):

```
(let x0 = \z->z in
  (let x1 = (x0,x0) in
    (let x2 = (x1,x1) in
      (let x3 = (x2,x2) in
        (let x4 = (x3,x3) in
          (let x5 = (x4,x4) in
            (let x6 = (x5,x5) in x6))))))))
```

Die Anzahl der Typvariablen ist  $64 = 2^6$ .

Verallgemeinert man das Beispiel mit Parameter  $n$ , dann sind  $2^n$   
Typvariablen notwendig.

# Beispiele: map

```
map f xs = case xs of {
    [] -> []
    (y:ys) -> (f y):(map f ys)
}
```

$$A = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a]\}$$

Sei  $A' = A \cup \{\text{map} :: \beta, f :: \alpha_1, xs :: \alpha_2\}$  und  $A'' = A' \cup \{y : \alpha_3, ys :: \alpha_4\}$ .

$$\begin{array}{l}
 (a) \quad A' \vdash xs :: \tau_1, E_1 \\
 (b) \quad A' \vdash \text{Nil} :: \tau_2, E_2 \\
 (c) \quad A'' \vdash (\text{Cons } y \text{ } ys) :: \tau_3, E_3 \\
 (d) \quad A' \vdash \text{Nil} :: \tau_4, E_4 \\
 (e) \quad A'' \vdash (\text{Cons } (f \text{ } y) (\text{map } f \text{ } ys)) :: \tau_5, E_5 \\
 \hline
 \text{(RCASE)} \quad \frac{}{A' \vdash \text{case } xs \text{ of } \{\text{Nil} \rightarrow \text{Nil}; \text{Cons } y \text{ } ys \rightarrow \text{Cons } y (\text{map } f \text{ } ys)\} :: \alpha, E} \\
 \text{(MSKREK1)} \quad \frac{}{A \vdash_T \text{map} :: \sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha)} \\
 \text{wenn } \sigma \text{ Lösung von } E \cup \{\beta \doteq \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha\}
 \end{array}$$

wobei  $E = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha \doteq \tau_4, \alpha \doteq \tau_5\}$ .

(a) bis (e) folgt

# Beispiele: map (2)

(a) 
$$\frac{(\text{AxV})}{A' \vdash xs :: \alpha_2, \emptyset}$$
  
 D.h.  $\tau_1 = \alpha_2$  und  $E_1 = \emptyset$ .

(b) 
$$\frac{(\text{AxK})}{A' \vdash \text{Nil} :: [\alpha_5], \emptyset}$$
  
 D.h.  $\tau_2 = [\alpha_5]$  und  $E_2 = \emptyset$

(c) 
$$\frac{\frac{(\text{AxK})}{A'' \vdash \text{Cons} :: \alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6]}, \frac{(\text{AxV})}{A'' \vdash y :: \alpha_3, \emptyset}}{(\text{RAPP}) \frac{A'' \vdash (\text{Cons } y) :: \alpha_7, \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \doteq \alpha_3 \rightarrow \alpha_7\}}{(\text{RAPP}) \frac{A'' \vdash (\text{Cons } y \text{ } ys) :: \alpha_8, \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \doteq \alpha_3 \rightarrow \alpha_7, \alpha_7 \doteq \alpha_4 \rightarrow \alpha_8\}}{A'' \vdash (\text{Cons } y \text{ } ys) :: \alpha_8, \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \doteq \alpha_3 \rightarrow \alpha_7, \alpha_7 \doteq \alpha_4 \rightarrow \alpha_8\}}}, \frac{(\text{AxV})}{A'' \vdash ys :: \alpha_4, \emptyset}$$
  
 D.h.  $\tau_3 = \alpha_8$  und  $E_3 = \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \doteq \alpha_3 \rightarrow \alpha_7, \alpha_7 \doteq \alpha_4 \rightarrow \alpha_8\}$

(d) 
$$\frac{(\text{AxK})}{A' \vdash \text{Nil} :: [\alpha_9], \emptyset}$$
  
 D.h.  $\tau_4 = [\alpha_9]$  und  $E_4 = \emptyset$ .

## Beispiele: map (3)

(e)

$$\frac{\frac{\frac{(\text{AxX})}{A'' \vdash \text{Cons} :: \alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}], \emptyset}, \frac{(\text{AxV})}{A'' \vdash f :: \alpha_1, \emptyset}, \frac{(\text{AxV})}{A'' \vdash y :: \alpha_3, \emptyset}}{(\text{RApp}) A'' \vdash (f y) :: \alpha_{15}, \{\alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}\}}, \frac{\frac{(\text{AxSK2})}{A'' \vdash \text{map} :: \beta, \emptyset}, \frac{(\text{AxV})}{A'' \vdash f :: \alpha_1, \emptyset}, \frac{(\text{AxV})}{A'' \vdash ys :: \alpha_4, \emptyset}}{(\text{RApp}) A'' \vdash (\text{map } f) :: \alpha_{12}, \{\beta \dot{=} \alpha_1 \rightarrow \alpha_{12}\}}, \frac{(\text{RApp}) A'' \vdash (\text{map } f \text{ } ys) :: \alpha_{13}, \{\beta \dot{=} \alpha_1 \rightarrow \alpha_{12}, \alpha_{12} \dot{=} \alpha_4 \rightarrow \alpha_{13}\}}{(\text{RApp}) A'' \vdash (\text{Cons } (f y) (\text{map } f \text{ } ys)) :: \alpha_{15}, \{\alpha_{11} \dot{=} \alpha_{13} \rightarrow \alpha_{14}, \alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \dot{=} \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}\}}$$

D.h.  $\tau_5 = \alpha_{14}$  und

$$E_5 = \{\alpha_{11} \dot{=} \alpha_{13} \rightarrow \alpha_{14}, \alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \dot{=} \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}, \\ \beta \dot{=} \alpha_1 \rightarrow \alpha_{12}, \alpha_{12} \dot{=} \alpha_4 \rightarrow \alpha_{13}\}$$

# Beispiele: map (4)

Gleichungssystem  $E \cup \{\beta \doteq \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha\}$  durch Unifikation lösen:

$$\begin{aligned} &\{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \doteq \alpha_3 \rightarrow \alpha_7, \alpha_7 \doteq \alpha_4 \rightarrow \alpha_8, \alpha_{11} \doteq \alpha_{13} \rightarrow \alpha_{14}, \\ &\alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \doteq \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \doteq \alpha_3 \rightarrow \alpha_{15}, \beta \doteq \alpha_1 \rightarrow \alpha_{12}, \\ &\alpha_{12} \doteq \alpha_4 \rightarrow \alpha_{13}, \alpha_2 \doteq [\alpha_5], \alpha_2 \doteq \alpha_8, \alpha \doteq [\alpha_9], \alpha \doteq \alpha_{14}, \\ &\beta \doteq \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha\} \end{aligned}$$

Die Unifikation ergibt

$$\begin{aligned} \sigma = \{ &\alpha \mapsto [\alpha_{10}], \alpha_1 \mapsto \alpha_6 \rightarrow \alpha_{10}, \alpha_2 \mapsto [\alpha_6], \alpha_3 \mapsto \alpha_6, \alpha_4 \mapsto [\alpha_6], \alpha_5 \mapsto \alpha_6, \\ &\alpha_7 \mapsto [\alpha_6] \rightarrow [\alpha_6], \alpha_8 \mapsto [\alpha_6], \alpha_9 \mapsto \alpha_{10}, \alpha_{11} \mapsto [\alpha_{10}] \rightarrow [\alpha_{10}], \\ &\alpha_{12} \mapsto [\alpha_6] \rightarrow [\alpha_{10}], \alpha_{13} \mapsto [\alpha_{10}], \alpha_{14} \mapsto [\alpha_{10}], \alpha_{15} \mapsto \alpha_{10}, \\ &\beta \mapsto (\alpha_6 \rightarrow \alpha_{10}) \rightarrow [\alpha_6] \rightarrow [\alpha_{10}], \end{aligned}$$

D.h.  $\text{map} :: \sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha) = (\alpha_6 \rightarrow \alpha_{10}) \rightarrow [\alpha_6] \rightarrow [\alpha_{10}]$ .

# Beispiele: erneute Betrachtung

$g \ x = x : (g \ (g \ 'c'))$

Iteratives Verfahren liefert Fail nach mehreren Iteration.

Hindley-Milner:  $A = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], 'c' :: \text{Char}\}$ .

Sei  $A' = A \cup \{x :: \alpha, g :: \beta\}$ .

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A' \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, \text{(AxV)} \frac{}{A' \vdash x :: \alpha, \emptyset} \quad \text{(AxSK2)} \frac{}{A' \vdash g :: \beta, \emptyset}, \text{(RApp)} \frac{}{A' \vdash (g \ 'c') :: \alpha_7, \{\beta \doteq \text{Char} \rightarrow \alpha_7\}} \\
 \text{(RApp)} \frac{}{A' \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha \rightarrow \alpha_3}, \text{(RApp)} \frac{}{A' \vdash (g \ (g \ 'c')) :: \alpha_4, \{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4\}} \\
 \text{(RApp)} \frac{}{A' \vdash \text{Cons } x \ (g \ (g \ 'c')) :: \alpha_2, \{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4 \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}} \\
 \text{(MSKREK)} \frac{}{A \vdash_T g :: \sigma(\alpha \rightarrow \alpha_2)} \\
 \text{wobei } \sigma \text{ die Lösung von}
 \end{array}$$

$\{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2, \beta \doteq \alpha \rightarrow \alpha_2\}$  ist.

Die Unifikation schlägt jedoch fehl, da Char mit einer Liste unifiziert werden soll. D.h. g ist nicht Hindley-Milner-typisierbar.

# Beispiele: erneute Betrachtung (2)

$g \ x = 1 : (g \ (g \ 'c'))$

Iteratives Verfahren liefert  $g :: \forall \alpha. \alpha \rightarrow [\text{Int}]$

Hindley-Milner: Sei  $A' = A \cup \{x :: \alpha, g :: \beta\}$ .

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A' \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, \text{(AxK)} \frac{}{A' \vdash 1 :: \text{Int}, \emptyset}, \text{(AxSK2)} \frac{}{A' \vdash g :: \beta, \emptyset}, \text{(RApP)} \frac{}{A' \vdash (g \ 'c') :: \alpha_7, \{\beta \doteq \text{Char} \rightarrow \alpha_7\}} \\
 \text{(RApP)} \frac{}{A' \vdash (\text{Cons } 1) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3}, \text{(RApP)} \frac{}{A' \vdash (g \ (g \ 'c')) :: \alpha_4, \{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4\}} \\
 \text{(RApP)} \frac{}{A' \vdash \text{Cons } 1 \ (g \ (g \ 'c')) :: \alpha_2, \{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}} \\
 \text{(SKREK)} \frac{}{A \vdash_T g :: \sigma(\alpha \rightarrow \alpha_2)}
 \end{array}$$

wobei  $\sigma$  die Lösung von

$\{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2, \beta \doteq \alpha \rightarrow \alpha_2\}$  ist.

Die Unifikation schlägt fehl, da  $[\alpha_5] \doteq \text{Char}$  unifiziert werden soll.

# Iteratives Verfahren kann allgemeinere Typen liefern

```
data Baum a = Leer | Knoten a (Baum a) (Baum a)
```

Die Typen für die Konstruktoren sind

`Leer` ::  $\forall a. \text{Baum } a$  und

`Knoten` ::  $\forall a. a \rightarrow \text{Baum } a \rightarrow \text{Baum } a \rightarrow \text{Baum } a$

```
g x y = Knoten True (g x y) (g y x)
```

Hindley-Milner-Typcheck  $g :: a \rightarrow a \rightarrow \text{Baum Bool}$

Iteratives Verfahren:  $g :: a \rightarrow b \rightarrow \text{Baum Bool}$

**Grund (im Verfahren):**

Iteratives Verfahren erlaubt Kopien des Typs für  $g$ , Hindley-Milner nicht.

Haskell akzeptiert für  $g$  die allgemeinere Typannahme:

```
g :: a -> b -> Baum Bool
```

```
g x y = Knoten True (g x y) (g y x)
```



# Potentielle Nutzung der iterativen Typisierung in Haskell



- Wenn Superkombinatoren  $S_1, \dots, S_n$  nicht Hindley-Milner-typisierbar oder Hindley-Milner liefert (vermutet) zu speziellen Typ
- Aktion: Typisiere  $S_1, \dots, S_n$  mit dem iterativen (polymorphen) Typisierungsverfahren
- Wenn Erfolg:  $S_1 : \tau_1, \dots, S_n : \tau_n$ ,
- Dann: spezifiziere im Programm-File :
- $s_1 :: \tau_1, \dots, s_n :: \tau_n$
- Danach kann Haskell die Typen und **verifizieren** und **verwenden** !

# Hindley-Milner Typisierung und Type Safety

- Hindley-Milner-getypte Programme sind immer auch iterativ typisierbar
- Daher sind Hindley-Milner getypte Programme niemals dynamisch ungetypt
- Es gilt auch das Progress-Lemma: Hindley-Milner getypte (geschlossene) Programme sind WHNFs oder reduzibel

# Hindley-Milner Typisierung und Type Safety (2)

- Type-Preservation: Gilt in KFPTSP+seq aber vermutlich nicht in Haskell (als Kernsprache mit let)

```
let x = (let y = \u -> z in (y [], y True, seq x True))  
      z = const z x  
in x
```

ist Hindley-Milner-typisierbar.

- Wenn man eine so genannte (*llet*)-Reduktion durchführt, erhält man:

```
let x = (y [], y True, seq x True)  
      y = \u -> z  
      z = const z x  
in x
```

Ist nicht mehr Hindley-Milner-typisierbar (in Kernsprache mit let)

# Hindley-Milner Typisierung und Type Safety (2)

- Type-Preservation: Gilt in  $KFPTSP+seq$  aber vermutlich nicht in Haskell (als Kernsprache mit `let`)

```
let x = (let y = \u -> z in (y [], y True, seq x True))
      z = const z x
in x
```

ist Hindley-Milner-typisierbar.

- Wenn man eine so genannte (*llet*)-Reduktion durchführt, erhält man:

```
let x = (y [], y True, seq x True)
      y = \u -> z
      z = const z x
in x
```

Ist nicht mehr Hindley-Milner-typisierbar (in Kernsprache mit `let`)

„Vermutlich“: Haskell's operationale Semantik ist anders definiert

# Hindley-Milner Typisierung und Type Safety (3)

Im Let-Kernsprache: Hindley-Milner-typisierbar:

```
let x = (let y = \u -> z in (y [], y True, seq x True))
      z = const z x
in x
```

- mittels (llet) reduziert das zu:

```
let x = (y [], y True, seq x True)
      y = \u -> z
      z = const z x
in x
```

NICHT Hindley-Milner typisierbar (aber iterativ typisierbar):

- Der Effekt kommt von der Allquantifizierung nach erfolgreicher Typisierung:

Vorher: einmal kann allquantifiziert werden

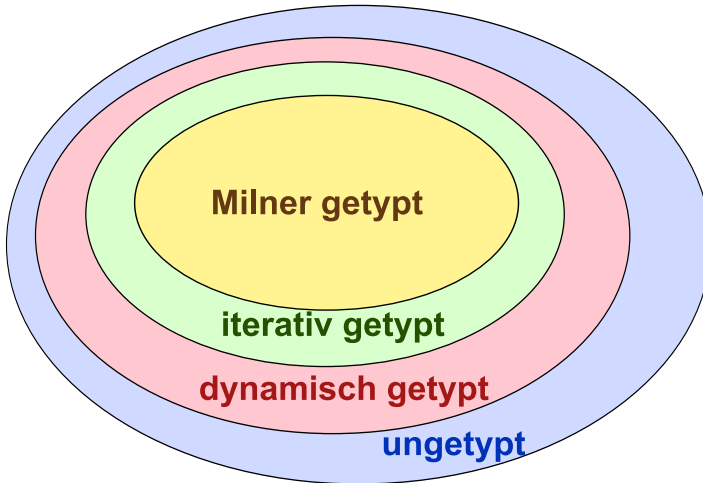
Nachher: alles wird auf einmal typisiert.

# Hindley-Milner Typisierung und Type Safety(4)

Das Beispiel ist aber unkritisch, denn:

- Type-Preservation gilt für das iterative Verfahren;
- typisierte Programme sind dynamisch getypt;
- Hindley-Milner-typisierbar impliziert iterativ typisierbar und
- Reduktion erhält iterative Typisierbarkeit

# KFPTS+seq



# Prädikativ / Imprädikativ

- **Prädikativer Polymorphismus:** Typvariablen stehen für Grundtypen (= Haskell, KFPTSP+seq)
- **Imprädikativer Polymorphismus:** Typvariablen stehen auch für polymorphe Typen (mit Quantoren!)



# Prädikativ / Imprädikativ

- **Prädikativer Polymorphismus:** Typvariablen stehen für Grundtypen (= Haskell, KFPTSP+seq)
- **Imprädikativer Polymorphismus:** Typvariablen stehen auch für polymorphe Typen (mit Quantoren!)

Versuch  $\lambda x \rightarrow \text{const } (x \text{ True}) (x \text{ 'A'})$  zu typisieren:

$x$  ist eine Funktion, die für alle Eingabetypen den gleichen Ergebnistyp liefert

# Prädikativ / Imprädikativ

- **Prädikativer Polymorphismus:** Typvariablen stehen für Grundtypen (= Haskell, KFPTSP+seq)
- **Imprädikativer Polymorphismus:** Typvariablen stehen auch für polymorphe Typen (mit Quantoren!)

Versuch `\x -> const (x True) (x 'A')` zu typisieren:

`x` ist eine Funktion, die für alle Eingabetypen den gleichen Ergebnistyp liefert

Mit imprädikativem Polymorphismus geht das:

```
(\x -> const (x True) (x 'A')) :: (forall b. (forall a. a -> b) -> b)
```

# Prädikativ / Imprädikativ

- **Prädikativer Polymorphismus:** Typvariablen stehen für Grundtypen (= Haskell, KFPTSP+seq)
- **Imprädikativer Polymorphismus:** Typvariablen stehen auch für polymorphe Typen (mit Quantoren!)

Versuch  $\lambda x \rightarrow \text{const } (x \text{ True}) (x \text{ 'A'})$  zu typisieren:

$x$  ist eine Funktion, die für alle Eingabetypen den gleichen Ergebnistyp liefert

Mit imprädikativem Polymorphismus geht das:

$(\lambda x \rightarrow \text{const } (x \text{ True}) (x \text{ 'A'})) :: (\text{forall } b. (\text{forall } a. a \rightarrow b) \rightarrow b)$

Aber:

- Kein Haskell, sondern Erweiterung
- Typinferenz / Typisierbarkeit nicht mehr entscheidbar!

# Typisierung unter Typklassen

## Typisierung unter Typklassen

# Typisierung unter Typklassen

## Annahmen:

- Erweiterung der Typisierungsalgorithmen auf Typklassen-Beschränkungen.
- Während der Typisierung kommen Typklassenbeschränkungen nur aus den Annahmen (Superkombinatoren)
- Basis: KFPTSP, erweitert um Typklassen.

## Beispiel

`genericLength :: Num b => [a] → b`

berechnet aus der Definition

Beschränkung kommt von der Addition `+`.

# Erweiterung um Typklassen

- Typklassen  $Cl$  als Namen
- Typen von Ausdrücken sind ein Paar, geschrieben:  $C \Rightarrow \tau$   
wobei  $C$  *Typklassenconstraint*,  $\tau$  ist polymorpher Typ.
- Ein Typklassenconstraint ist eine Menge von Ausdrücken  $Cl\ a$ .  
 $Cl$  ist ein Typklassenname und  $a$  eine Typvariable.  
(Alle Typvariablen in  $C$  kommen auch in  $\tau$  vor.)
- Es gibt vorgegebene Funktionen (auch *Klassenfunktionen*)  
deren Typ schon nichttriviale Typklassenconstraints enthält.  
(Haskell: Konstruktoren sind ohne Typklassenconstraints)

## Beispiel

- `Num` ist Typklasse der (Basis-)Typen zu Zahlen
- `(+) :: Num a => a -> a -> a`

Eine global vorgegebene Menge  $M_{\text{Typklassenaxiome}}$  von Formeln:

- ①  $Cl\ \tau$  (d.h.  $\tau \in Cl$ ) für Basistypen  $\tau$ .  
: z.B.:  $Cl(\text{List Bool})$  ist nicht möglich
- ② Implikationen der Form  $C \implies Cl(TC\ a_1 \dots a_n)$   
wobei  $a_1, \dots, a_n$  verschiedene Typvariablen sind und in  $C$  nur Typklassenconstraints der Form  $Cl_i\ a_i$  vorkommen.  
Pro Typkonstruktor  $TC$  darf es nur eine solche Implikation geben.
- ③ Implikationen der Form  $Cl_1\ a \implies Cl_2\ a$ .

# Berechnungen auf Typklassenconstraints

Fragestellung: gehört ein (Grund-) Typ zu einer Typklasse?

## Verfahren: Typklassenconstraints entscheiden

Eingabe: Constraint-Menge  $\mathcal{C} = \{Cl \ \tau\}$ .

Vereinfache die Menge mit den zwei folgenden Schritten solange, bis die Menge leer ist und somit alle Constraints erfüllt.

- ① Wähle ein Constraint  $Cl \ TC$  aus  $\mathcal{C}$ : wenn dies gilt, d.h. in  $M_{Typklassenaxiome}$  enthalten ist, wobei wir die einfachen Implikation  $Cl_1 \ a \implies Cl_2 \ a$  hierbei mitberücksichtigen, dann entferne das Constraint aus der Menge  $\mathcal{C}$ .
- ② Nehme ein Constraint  $Cl \ (TC \ \tau_1 \dots \tau_n)$  aus  $\mathcal{C}$  (mit maximaler Größe); wenn es eine Implikation  $C_0 \implies Cl(TC \ a_1 \dots a_n)$  gibt, dann ersetze das Constraint in  $\mathcal{C}$  durch die Menge  $\sigma(C_0)$ , wobei  $\sigma = \{a_1 \mapsto \tau_1, \dots, a_n \mapsto \tau_n\}$  ist.

Wenn die Constraint-Menge leer ist, dann ergibt sich True.

Wenn diese nicht komplett eliminiert werden kann, ergibt sich insgesamt False.



# Typklassenconstraints: Beispiel

Gegeben als Axiome:

- $\text{Eq Int, und Eq Bool,}$
- $\text{Eq } a \implies \text{Eq } [a].$
- $\{\text{Eq } a, \text{Eq } b\} \implies \text{Eq } (a, b).$

Gilt  $\text{Eq } ([\text{Int}], \text{Bool}). ?$

# Typklassenconstraints: Beispiel

Gegeben als Axiome:

- $\text{Eq Int}$ , und  $\text{Eq Bool}$ ,
- $\text{Eq } a \implies \text{Eq } [a]$ .
- $\{\text{Eq } a, \text{Eq } b\} \implies \text{Eq } (a, b)$ .

Gilt  $\text{Eq } ([\text{Int}], \text{Bool})$ . ?

Start:  $\{\text{Eq } ([\text{Int}], \text{Bool})\}$

# Typklassenconstraints: Beispiel

Gegeben als Axiome:

- $\text{Eq Int}$ , und  $\text{Eq Bool}$ ,
- $\text{Eq } a \implies \text{Eq } [a]$ .
- $\{\text{Eq } a, \text{Eq } b\} \implies \text{Eq } (a, b)$ .

Gilt  $\text{Eq } ([\text{Int}], \text{Bool})$ . ?

Start:  $\{\text{Eq } ([\text{Int}], \text{Bool})\}$   
 $\{\text{Eq } [\text{Int}], \text{Eq Bool}\}$  wg. Implikation für Paare

# Typklassenconstraints: Beispiel

Gegeben als Axiome:

- $\text{Eq Int, und Eq Bool,}$
- $\text{Eq } a \implies \text{Eq } [a].$
- $\{\text{Eq } a, \text{Eq } b\} \implies \text{Eq } (a, b).$

Gilt  $\text{Eq } ([\text{Int}], \text{Bool}). ?$

Start:

|                                               |                            |
|-----------------------------------------------|----------------------------|
| $\{\text{Eq } ([\text{Int}], \text{Bool})\}$  |                            |
| $\{\text{Eq } [\text{Int}], \text{Eq Bool}\}$ | wg. Implikation für Paare  |
| $\{\text{Eq Int, Eq Bool}\}$                  | wg. Implikation für Listen |

# Typklassenconstraints: Beispiel

Gegeben als Axiome:

- $\text{Eq Int}$ , und  $\text{Eq Bool}$ ,
- $\text{Eq } a \implies \text{Eq } [a]$ .
- $\{\text{Eq } a, \text{Eq } b\} \implies \text{Eq } (a, b)$ .

Gilt  $\text{Eq } ([\text{Int}], \text{Bool})$ . ?

|           |                                               |                            |
|-----------|-----------------------------------------------|----------------------------|
| Start:    | $\{\text{Eq } ([\text{Int}], \text{Bool})\}$  |                            |
|           | $\{\text{Eq } [\text{Int}], \text{Eq Bool}\}$ | wg. Implikation für Paare  |
|           | $\{\text{Eq Int}, \text{Eq Bool}\}$           | wg. Implikation für Listen |
|           | $\emptyset$ ,                                 | da beide gelten            |
| Ergebnis: | True                                          |                            |

# Was ist ein Baumautomat?

- Analog zu endlichem Automaten, aber statt Strings werden endliche Bäume eingelesen und akzeptiert oder verworfen.
- Bäume sind first-order Terme ohne Variablen.  
Werden manchmal auch „ranked trees“ genannt.

Ein Baum  $B$  wird folgendermaßen verarbeitet von einem Baumautomaten  $T$ :

- 1 Jedes Blatt wird mit dem Zustand entsprechend  $T$  markiert
- 2 Knoten werden markiert (von den Blättern her):  
Wenn  $f\ s_1 \dots s_n$  der Knoten ist,  
und  $s_i$  schon mit  $a_i$  markiert ist, dann markiere Knoten mit dem label  $f(a_1, \dots, a_n)$  entsprechend dem Baumautomaten  $T$
- 3 Wenn die Wurzel mit  $a$  markiert wird, und  $a$  ist ein akzeptierender Zustand von  $T$ , dann wird der Baum  $B$  von  $T$  akzeptiert.
- 4 Die Menge der akzeptierten Bäume ist die Baumsprache zu  $T$ .

## Aussagenlogische Auswertung

. . . . .

- Das algorithmische Problem, ob  $Cl \tau$  gilt, ist eigentlich dasselbe wie die Frage, ob ein **Baumautomat** einen gegebenen Baum **akzeptiert** oder nicht.
- Der Baumautomat ist gegeben durch die Typklassenaxiome.
- Hier gibt es den Unterschied, ob der Baumautomat deterministisch ist oder nicht, und ob er von oben oder von unten den Baum abarbeitet.
- Das Problem ist mit einem **polynomiellen Algorithmus** entscheidbar.
- Zu allgemeinen Aussagen, insbesondere Komplexität, siehe Buch zu Tree Automata (Literatur im Skript). In speziellen Fall der Typklassen hat man deterministische Varianten.



**Definition**

Die (Grundtypen-)Semantik eines Typs unter den Constraints kann man so definieren:

$$\begin{aligned} \text{sem}(C, \tau) = \{ & \sigma(\tau) \mid \sigma \text{ setzt Grundtypen für Typvariablen ein} \\ & \text{und für alle Constraints} \\ & (TC \ a) \in C : (\sigma(a) \in \text{sem}(TC)) \} \end{aligned}$$

$\Rightarrow$  Die Axiome kann man als Mengendefinitionen ansehen.

# Typklassen: Beispiele für Typisierung

Typklassen, Axiome und Klassenfunktionen zu  
Haskell-Typklassen wie `Num`, `Ord`, und `Show` sind vorhanden.  
Ebenso Typ von Klassenfunktionen wie `+` ist schon gegeben als:  
$$+ :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a.$$

# Typklassen: Beispiele für Typisierung

Typklassen, Axiome und Klassenfunktionen zu Haskell-Typklassen wie `Num`, `Ord`, und `Show` sind vorhanden. Ebenso `Typ` von Klassenfunktionen wie `+` ist schon gegeben als:

$$+ :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a.$$

## Beispiel Typisierung

$$\lambda x. \lambda y. (x, y, x + y).$$

# Typklassen: Beispiele für Typisierung

Typklassen, Axiome und Klassenfunktionen zu Haskell-Typklassen wie `Num`, `Ord`, und `Show` sind vorhanden. Ebenso Typ von Klassenfunktionen wie `+` ist schon gegeben als:

$$+ :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a.$$

## Beispiel Typisierung

$$\lambda x. \lambda y. (x, y, x + y).$$
$$x :: \alpha_1, y :: \alpha_2$$

Typ von `+` erzwingt:  $a = \alpha_1 = \alpha_2$  und `Num a`.

# Typklassen: Beispiele für Typisierung

Typklassen, Axiome und Klassenfunktionen zu Haskell-Typklassen wie `Num`, `Ord`, und `Show` sind vorhanden. Ebenso Typ von Klassenfunktionen wie `+` ist schon gegeben als:

$$+ :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a.$$

## Beispiel Typisierung

$$\lambda x. \lambda y. (x, y, x + y).$$
$$x :: \alpha_1, y :: \alpha_2$$

Typ von `+` erzwingt:  $a = \alpha_1 = \alpha_2$  und `Num a`.

Es ergibt sich:

$$\lambda x. \lambda y. (x, y, x + y) :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow (a, a, a)$$

# Typisierung unter Typklassen: Unifikation

## Änderungen der Unifikation:

- 1 Man startet mit einer Gleichung  $s \doteq t$  und eine Menge  $\mathcal{C}$  von Typklassenconstraints für Typvariablen.
- 2 Man wendet die Unifikationsregeln auf die Gleichungen an, bis sich am Ende eine Substitution  $\sigma$  ergibt.
- 3 Man vereinfacht die Constraint-Menge vollständig.  
Wenn danach alle Constraints nur noch die Form  $(TC\ a)$  haben, dann ist das Verfahren erfolgreich.  
Ergebnis ist die Substitution  $\sigma$  und das Constraint  $\mathcal{C}'$  als  $\sigma\mathcal{C}$ .

# Typisierung unter Typklassen: Unifikation

- Analog sind die Änderungen bei den Typisierungsverfahren von Ausdrücken und Superkombinatoren.
- Dies gilt für das Hindley-Milner-Typisierungsverfahren.
- Iteratives Typisierungsverfahren: sollte auch gehen:  
es werden nicht nur in jedem Schritt die Typen verfeinert,  
sondern auch die Constraints.

# Typisierung unter Typklassen: Beispiel

Addition auf Paaren:  $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$

Implikationsaxiom für Paare:  $\{\text{Num } a, \text{Num } b\} \implies \text{Num } (a, b)$ .

Typisiere die Funktion:  $f\ x = x + (1, 2)$



# Typisierung unter Typklassen: Beispiel

Addition auf Paaren:  $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$

Implikationsaxiom für Paare:  $\{\text{Num } a, \text{Num } b\} \implies \text{Num } (a, b)$ .

Typisiere die Funktion:  $f \ x = x + (1, 2)$

$x :: \alpha$ .

$+ :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a$ .

Gleichungen:  $\alpha \doteq a, a \doteq (\text{Int}, \text{Int})$

Constraint:  $\{\text{Num } a\}$ .

# Typisierung unter Typklassen: Beispiel

Addition auf Paaren:  $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$

Implikationsaxiom für Paare:  $\{\text{Num } a, \text{Num } b\} \implies \text{Num } (a, b)$ .

Typisiere die Funktion:  $f \ x = x + (1, 2)$

$x :: \alpha$ .

$+ :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a$ .

Gleichungen:  $\alpha \doteq a, a \doteq (\text{Int}, \text{Int})$

Constraint:  $\{\text{Num } a\}$ .

Unifikation ergibt  $\sigma = \{a \mapsto (\text{Int}, \text{Int}), \alpha \mapsto (\text{Int}, \text{Int})\}$

Constraintmenge:  $\{\text{Num } (\text{Int}, \text{Int})\}$ .

# Typisierung unter Typklassen: Beispiel

Addition auf Paaren:  $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$

Implikationsaxiom für Paare:  $\{\text{Num } a, \text{Num } b\} \implies \text{Num } (a, b)$ .

Typisiere die Funktion:  $f \ x = x + (1, 2)$

$x :: \alpha$ .

$+ :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a$ .

Gleichungen:  $\alpha \doteq a, a \doteq (\text{Int}, \text{Int})$

Constraint:  $\{\text{Num } a\}$ .

Unifikation ergibt  $\sigma = \{a \mapsto (\text{Int}, \text{Int}), \alpha \mapsto (\text{Int}, \text{Int})\}$

Constraintmenge:  $\{\text{Num } (\text{Int}, \text{Int})\}$ .

Implikation und Basisconstraints zeigen, dass das Constraint gilt!

Resultat:  $f :: (\text{Int}, \text{Int}) \rightarrow (\text{Int}, \text{Int})$

# Typisierung unter Typklassen: Beispiel

Typisiere die Funktion:  $g\ x\ y = x + (y, y)$

# Typisierung unter Typklassen: Beispiel

Typisiere die Funktion:  $g\ x\ y = x + (y, y)$

$x :: \alpha, y :: \beta$

$+ :: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a.$

Gleichungen:  $\alpha \doteq (\beta, \beta), a \doteq \alpha$

Constraint:  $\{\text{Num } a\}.$

# Typisierung unter Typklassen: Beispiel

Typisiere die Funktion:  $g\ x\ y = x + (y, y)$

$x :: \alpha, y :: \beta$

$+$   $:: \{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a.$

Gleichungen:  $\alpha \doteq (\beta, \beta), a \doteq \alpha$

Constraint:  $\{\text{Num } a\}.$

Unifikation ergibt  $\sigma = \{a \mapsto (\beta, \beta), \alpha \mapsto (\beta, \beta)\}$

Constraintmenge:  $\{\text{Num } (\beta, \beta)\}.$

# Typisierung unter Typklassen: Beispiel

Typisiere die Funktion:  $g\ x\ y = x + (y, y)$

$x :: \alpha, y :: \beta$

$+$  ::  $\{\text{Num } a\} \Rightarrow a \rightarrow a \rightarrow a$ .

Gleichungen:  $\alpha \doteq (\beta, \beta), a \doteq \alpha$

Constraint:  $\{\text{Num } a\}$ .

Unifikation ergibt  $\sigma = \{a \mapsto (\beta, \beta), \alpha \mapsto (\beta, \beta)\}$

Constraintmenge:  $\{\text{Num } (\beta, \beta)\}$ .

Implikation ergibt als Constraint:  $\{\text{Num } \beta\}$ .

Resultat:  $g :: \{\text{Num } \beta\} \Longrightarrow (\beta, \beta) \rightarrow \beta \rightarrow (\beta, \beta)$

# Typisierung unter Typklassen: Beispiel

```
genericLength xs = case xs of [] -> 0;
                        y:ys -> 1 + genericLength ys
```

Typisierung; ergibt Gleichungen und Bedingungen:

$$xs :: \alpha_1, 0 :: \text{Num } b_1 \implies b_1, 1 :: \text{Num } b_2 \implies b_2$$

$$\text{genericLength} :: \alpha_1 \rightarrow \alpha_2.$$

$$\text{wegen } (\text{case } xs \text{ of } [] \dots): \alpha_1 = [\alpha_4]$$

$$\text{genericLength } ys :: \alpha_2$$

$$+ :: \text{Num } a \implies a \rightarrow a \rightarrow a.$$

$$a = b_2, a = \alpha_2, b_1 = \alpha_2; \text{Num } a, \text{Num } b_1, \text{Num } b_2.$$

Ergibt insgesamt:  $\text{genericLength} :: \text{Num } a \implies [b] \rightarrow a$