GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Einführung in die Funktionale Programmierung:

Typisierung

Teil 2

Prof Dr. Manfred Schmidt-Schauß

WS 2023/24

Stand der Folien: 16. Januar 2024

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.
- $SK_1, \ldots, SK_m$ sind verschränkt rekursiv, wenn $SK_i \preceq^+ SK_j$ für alle $i, j \in \{1, \ldots, m\}$

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 τ typisieren

Typisierungsregel für (geschlossene) nicht-rekursive SK:

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

wenn σ Lösung von E,

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

und SK nicht rekursiv ist,

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

τ-Notation: τ steht für einen Typ innerhalb der Berechnung

Beispiel: Typisierung von (.)

```
(.) f g x = f (g x)
```

A_0 ist leer, da keine Konstruktoren oder SK vorkommen.

$$\dfrac{\dfrac{\overset{\text{(AxV)}}{A_1 \vdash f :: \alpha_1, \emptyset}\ ,\ \overset{\text{(RApp)}}{\dfrac{\overset{\text{(AxV)}}{A_1 \vdash g :: \alpha_2, \emptyset}\ ,\ \overset{\text{(AxV)}}{A_1 \vdash x :: \alpha_3, \emptyset}}{A_1 \vdash (g\ x) :: \alpha_5, \{\alpha_2 \doteq \alpha_3 \to \alpha_5\}}}}{\overset{\text{(RApp)}}{A_1 \vdash (f\ (g\ x)) :: \alpha_4, \{\alpha_2 \doteq \alpha_3 \to \alpha_5, \alpha_1 = \alpha_5 \to \alpha_4\}}}}{\overset{\text{(RSK1)}}{\emptyset \vdash_T (.) :: \forall \mathcal{X}.\sigma(\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4)}}$$

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

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

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

$$(.) :: \forall a, b, c.(a \to b) \to (c \to a) \to c \to b$$

Typisierung von rekursiven Superkombinatoren

- Sei $SK\ x_1\ \ldots\ x_n = e$
- 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 $\mathrm{sem}(T) = \{\text{alle Grundtypen}\}$.
Das liefert der Typ α (bzw. quantifiziert $\forall \alpha.\alpha$)

Iteratives Typisierungsverfahren

Regel zur Berechnung neuer Annahmen:

$$(\text{SKR}_{\text{EK}})\ \frac{A \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \vdash_T SK :: \sigma(\alpha_1 \to \ldots \alpha_n \to \tau)}$$

wenn $SK\ x_1\ \ldots\ x_n = s$ die Definition von SK, σ 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

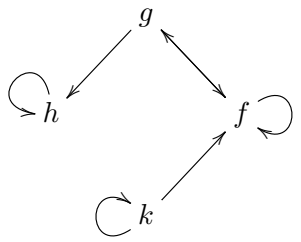
Typisierung der Gruppen entsprechend der $\preceq^+$-Ordnung modulo $\simeq$.

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

1. Anfangsannahme A enthält Typen der Konstruktoren der bereits bekannten Superkombinatoren
2. $A_0 := A \cup \{SK_1 :: \forall\alpha_1.\alpha_1, \ldots, SK_m :: \forall\alpha_m.\alpha_m\}$ und $j = 0$.
3. Verwende für jeden Superkombinator SK_i (mit $i = 1, \ldots, m$) 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: $SK_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SK_m :: \forall\mathcal{X}_m.\tau_m$
   Setze $A_{j+1} := A \cup \{SK_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SK_m :: \forall\mathcal{X}_m.\tau_m\}$
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 $SK_1, \ldots, SK_m$ nicht typisierbar.

Eigenschaften des Algorithmus

- Die berechneten Typen pro Iterationsschritt sind eindeutig bis auf Umbenennung.
  $\Longrightarrow$ bei Terminierung liefert der Algorithmus eindeutige Typen.
- Pro Iteration werden die neuen Typen spezieller (oder bleiben gleich).
  D.h. Monotonie bzgl. der Grundtypensemantik:
  $\mathrm{sem}(T_j) \supseteq \mathrm{sem}(T_{j+1})$
- 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)

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

Annahme:

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

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

$$\dfrac{\begin{array}{ll}(a) & A_0 \cup \{xs :: \alpha_1\} \vdash xs :: \tau_1, E_1\\ (b) & A_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: \tau_2, E_2\\ (c) & A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (y : ys) :: \tau_3, E_3\\ (d) & A_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \tau_4, E_4\\ (e) & A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}\} \vdash (1 + \texttt{length}\ ys) :: \tau_5, E_5\end{array}}{\dfrac{A_0 \cup \{xs :: \alpha_1\} \vdash (\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ xs\}) :: \alpha_3,\ E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha_3 \doteq \tau_4, \alpha_3 \doteq \tau_5\}}{A_0 \vdash_T \texttt{length} :: \sigma(\alpha_1 \to \alpha_3)}\ (\text{SKRek})}\ (\text{RCase})$$

wobei σ Lösung von

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

Beispiele: length (2)

(a): $\dfrac{}{A_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset}\ (\text{AxV})$
D.h $\tau_1 = \alpha_1$ und $E_1 = \emptyset$

(b): $\dfrac{}{A_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_6], \emptyset}\ (\text{AxK})$
D.h. $\tau_2 = [\alpha_6]$ und $E_2 = \emptyset$

(c)
$$\dfrac{\dfrac{\dfrac{}{A_0' \vdash (:) :: \alpha_9 \to [\alpha_9] \to [\alpha_9], \emptyset}\ (\text{AxK}) \quad , \quad \dfrac{}{A_0' \vdash y :: \alpha_4, \emptyset}\ (\text{AxV})}{A_0' \vdash ((:)\ y) :: \alpha_8, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8\}}\ (\text{RApp}) \quad , \quad \dfrac{}{A_0' \vdash ys :: \alpha_5, \emptyset}\ (\text{AxV})}{A_0' \vdash (y : ys) :: \alpha_7, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}}\ (\text{RApp})$$

wobei $A_0 = A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$
D.h. $\tau_3 = \alpha_7$ und $E_3 = \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}$

Beispiele: length (3)

(d) $\dfrac{(\text{AxK})}{A_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \texttt{Int}, \emptyset}$

D.h. $\tau_4 = \texttt{Int}$ und $E_4 = \emptyset$

(e)
$$\dfrac{\dfrac{\dfrac{}{A_0' \vdash (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}, \emptyset}(\text{AxK}) \quad \dfrac{}{A_0' \vdash 1 :: \texttt{Int}, \emptyset}(\text{AxK})}{A_0' \vdash ((+)\ 1) :: \alpha_{11}, \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}\}}(\text{RApp}) \quad \dfrac{}{A_0' \vdash \texttt{length} :: \alpha_{13}, \emptyset}(\text{AxSK}) \quad \dfrac{}{A_0' \vdash (ys) :: \alpha_5, \emptyset}(\text{AxV})}{A_0' \vdash (1 + \texttt{length}\ ys) :: \alpha_{10}, \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}}(\text{RApp})(\text{RApp})$$

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

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

$E_5 = \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}$

Beispiele: length (3)

Zusammengefasst:

$$A_0 \vdash_T \texttt{length} :: \sigma(\alpha_1 \to \alpha_3)$$

wobei σ Lösung von

$\{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7,$
$\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10},$
$\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\}$

Die Unifikation ergibt als Unifikator

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

daher $\sigma(\alpha_1 \to \alpha_3) = [\alpha_9] \to \texttt{Int}$

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

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 Typiserung von length, vereinfacht

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

1.Iteration: $A_0 = A \cup \{\texttt{length} :: \forall\alpha.\alpha\}$
bekannt: $\{\texttt{Nil} :: \forall a.[a], (:) :: \forall a.a \to [a] \to [a], 0, 1 :: \texttt{Int}, (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}\}$

Bedingungen ergeben:

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

Iterative Typiserung von length, vereinfacht

Zweiter Schritt: mit $\texttt{length} :: (\forall\alpha : [\alpha] \to \texttt{Int})$ im Rumpf

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

2.Iteration: $A_1 = A \cup \{\texttt{length} :: \forall\alpha.[\alpha] \to \texttt{Int}\}$
bekannt: $\{\texttt{Nil} :: \forall a.[a], (:) :: \forall a.a \to [a] \to [a], 0, 1 :: \texttt{Int}, (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}\}$

Bedingungen ergeben:

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

Iteratives Verfahren ist allgemeiner als Haskell

Beispiel

```
g x = 1 : (g (g 'c'))
```

$A = \{1 :: \texttt{Int}, \texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$
$A_0 = A \cup \{\texttt{g} :: \forall \alpha.\alpha\}$ (und $A_0' = A_0 \cup \{x :: \alpha_1\}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_0' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}(\text{AxK}) \quad \dfrac{}{A_0' \vdash 1 :: \texttt{Int}, \emptyset}(\text{AxK})}{A_0' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}(\text{RApp}) \quad \dfrac{\dfrac{}{A_0' \vdash \texttt{g} :: \alpha_6, \emptyset}(\text{AxSK}) \quad \dfrac{\dfrac{\dfrac{}{A_0' \vdash \texttt{g} :: \alpha_8, \emptyset}(\text{AxSK}) \quad \dfrac{}{A_0' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}(\text{AxK})}{A_0' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7\}}(\text{RApp})}{A_0' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}(\text{RApp})}{A_0' \vdash \texttt{Cons}\ 1\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}(\text{RApp})}{A_0 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_1 \to [\texttt{Int}]}(\text{SKRek})$$

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

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

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'))
```

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)

```
g x = x : (g (g 'c'))
```

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_0' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}(\text{AxK}) \quad \dfrac{}{A_0' \vdash x :: \alpha_1, \emptyset}(\text{AxV})}{A_0' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}(\text{RApp}) \quad \dfrac{\dfrac{}{A_0' \vdash \texttt{g} :: \alpha_6, \emptyset}(\text{AxSK}) \quad \dfrac{\dfrac{\dfrac{}{A_0' \vdash \texttt{g} :: \alpha_8, \emptyset}(\text{AxSK}) \quad \dfrac{}{A_0' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}(\text{AxK})}{A_0' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7\}}(\text{RApp})}{A_0' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}(\text{RApp})}{A_0' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}(\text{RApp})}{A_0 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_5 \to [\alpha_5]}(\text{SKRek})$$

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

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

Bsp.: Mehrere Iterationen sind nötig (2)

Da $A_0 \neq A_1$ muss eine weitere Iteration durchgeführt werden.
mit $\texttt{g} :: \forall \alpha.\alpha \to [\alpha]$

```
g x = x : (g (g 'c'))
```

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_1' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}(\text{AxK}) \quad \dfrac{}{A_1' \vdash x :: \alpha_1, \emptyset}(\text{AxV})}{A_1' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}(\text{RApp}) \quad \dfrac{\dfrac{}{A_1' \vdash \texttt{g} :: \alpha_6 \to [\alpha_6], \emptyset}(\text{AxSK}) \quad \dfrac{\dfrac{\dfrac{}{A_1' \vdash \texttt{g} :: \alpha_8 \to [\alpha_8], \emptyset}(\text{AxSK}) \quad \dfrac{}{A_1' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}(\text{AxK})}{A_1' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7\}}(\text{RApp})}{A_1' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4\}}(\text{RApp})}{A_1' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}(\text{RApp})}{A_1 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = [\texttt{Char}] \to [[\texttt{Char}]]}(\text{SKRek})$$

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

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

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 $A_2' = A_2 \cup \{x :: \alpha_1\} \quad = A \cup \{\texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], x :: \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{(\text{AxK})\ \overline{A_2' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\ ,\ (\text{AxV})\ \overline{A_2' \vdash x :: \alpha_1, \emptyset}}{(\text{RApp})\ A_2' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}\ ,\ (\text{AxSK})\ \overline{A_2' \vdash \texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], \emptyset}}{(\text{RApp})\ A_2' \vdash (\texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'}))) :: \alpha_2, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\ ,\ \dfrac{(\text{AxSK})\ \overline{A_2' \vdash \texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], \emptyset}\ ,\ (\text{AxK})\ \overline{A_2' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}}{(\text{RApp})\ A_2' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7\}}}{(\text{RApp})\ A_2' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4\}}}{(\text{SKRek})\ A_2' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}$$

$$A_2 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2)$$

wobei σ die Lösung von

$\{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ ist.

Unifikation:

$$\dfrac{\dfrac{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7,}{\dots}}{\dfrac{[\texttt{Char}] \doteq \texttt{Char},}{\dfrac{[[\texttt{Char}]] \doteq \alpha_7,}{\dots}}}$$

$$Fail$$

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 $\texttt{f} \simeq \texttt{g}$, d.h. das iterative Verfahren typisiert f und g gemeinsam.
$A = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} : \forall a.a\}$.
$A_0 = A \cup \{\texttt{f} :: \forall \alpha.\alpha, \texttt{g} :: \forall \alpha.\alpha\}$

$$\dfrac{\dfrac{\dfrac{(\text{AxK})\ \overline{A_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\ ,\ (\text{AxSK})\ \overline{A_0 \vdash \texttt{g} :: \alpha_5}}{(\text{RApp})\ A_0 \vdash (\texttt{Cons}\ \texttt{g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}\ ,\ (\text{AxK})\ \overline{A_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(\text{RApp})\ A_0 \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}{(\text{SKRek})\ A_0 \vdash_T \texttt{f} :: \sigma(\alpha_1) = [\alpha_5]}$$

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

Nichtterminierung des iterativen Verfahrens (2)

$$\dfrac{\dfrac{\dfrac{(\text{AxK})\ \overline{A_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\ ,\ (\text{AxSK})\ \overline{A_0 \vdash \texttt{f} :: \alpha_5}}{(\text{RApp})\ A_0 \vdash (\texttt{Cons}\ \texttt{f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}\ ,\ (\text{AxK})\ \overline{A_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(\text{RApp})\ A_0 \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}{(\text{SKRek})\ A_0 \vdash_T \texttt{g} :: \sigma(\alpha_1) = [\alpha_5]}$$

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

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

Nichtterminierung des iterativen Verfahrens (3)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{A_1 \vdash \texttt{g} :: [\alpha_5]}\,(\text{AxSK})}{A_1 \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{A_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{A_1 \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{A_1 \vdash_T \texttt{f} :: \sigma(\alpha_1) = [[\alpha_5]]}\,(\text{SKRek})$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{A_1 \vdash \texttt{f} :: [\alpha_5]}\,(\text{AxSK})}{A_1 \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{A_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{A_1 \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{A_1 \vdash_T \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]]}\,(\text{SKRek})$$

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

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

Nichtterminierung des iterativen Verfahrens (4)

Vermutung: Terminiert nicht

Beweis: (Induktion) betrachte den i. Schritt:

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{A_i \vdash \texttt{g} :: [\alpha_5]^i}\,(\text{AxSK})}{A_i \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{A_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{A_i \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{A_i \vdash_T \texttt{f} :: \sigma(\alpha_1) = [[\alpha_5]^i]}\,(\text{SKRek})$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{A_i \vdash \texttt{f} :: [\alpha_5]^i}\,(\text{AxSK})}{A_i \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{A_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{A_i \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{A_i \vdash_T \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]^i]}\,(\text{SKRek})$$

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

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

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 τ:
  Wenn $t :: \tau$ vor der Reduktion $t \to t'$,
  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.:

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

Hindley-Milner Typisierung

Roger Hindley; Robin Milner und Luis Damas haben beigetragen.

Erzwingen der Terminierung (der Typcheck-Iteration)

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

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

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

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

Erzwingen der Terminierung (2)

$$(\text{SKRekM}) \quad \frac{\text{für } i = 1, \ldots, m: \quad A_M \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i', E_i}{A_M \vdash_T \text{für } i = 1, \ldots, m \; SK_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i')}$$

wenn σ Lösung von $E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\tau_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i'\}$

und $SK_1 \; x_{1,1} \; \ldots \; x_{1,n_1} \;=\; s_1$

$\ldots$

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

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

Als zusätzliche Regel muss im Typisierungsverfahren hinzugefügt werden:

$$(\text{AxSK2}) \quad \frac{}{A \cup \{SK :: \tau\} \vdash SK :: \tau}$$

wenn τ nicht allquantifiziert ist

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 α_i (ohne Allquantor) in den Annahmen.
- Hindley-Milner typiserbar 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:
$SK_1, \ldots, SK_m$ 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 $SK_1, \ldots, SK_m$ mit der Regel (MSKREK):

$$(\mathrm{MSKREK})\ \frac{\begin{array}{c}\text{für } i = 1, \ldots, m: \\ A \cup \{SK_1 :: \beta_1, \ldots, SK_m :: \beta_m\} \\ \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i, E_i\end{array}}{A \vdash_T \text{für } i = 1, \ldots, m\ SK_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i)}$$

wenn σ Lösung von $E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\beta_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i\}$

und $\begin{array}{lcl} SK_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = & s_1 \\ \ldots & & \\ SK_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = & s_m \end{array}$

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

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

Das Hindley-Milner-Typisierungsverfahren (2)

Vereinfachung: Regel für einen (unabhängigen) rekursiven SK

$$(\mathrm{MSKREK1})\ \frac{A \cup \{SK :: \beta, x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \vdash_T SK :: \sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

wenn σ Lösung von $E \cup \{\beta \doteq \alpha_1 \to \ldots \to \alpha_n \to \tau\}$
und $SK\ x_1\ \ldots\ x_n = s$ die Definition von SK ist

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)
           }
```

$A = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a]\}$

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

$$\text{(MSKRek1)}\ \frac{\begin{array}{l}(a)\quad A' \vdash xs :: \tau_1, E_1\\ (b)\quad A' \vdash \texttt{Nil} :: \tau_2, E_2\\ (c)\quad A'' \vdash (\texttt{Cons}\ y\ ys) :: \tau_3, E_3\\ (d)\quad A' \vdash \texttt{Nil} :: \tau_4, E_4\\ (e)\quad A'' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \tau_5, E_5\end{array}}{\begin{array}{c}A' \vdash \texttt{case}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; \texttt{Cons}\ y\ ys \to \texttt{Cons}\ y\ (\texttt{map}\ f\ ys)\} :: \alpha, E\\ \hline A \vdash_T \texttt{map} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha)\\ \text{wenn } \sigma \text{ Lösung von } E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}\end{array}}\ \text{(RCase)}$$

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

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

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

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

Beispiele: map (3)

(e)

$$\dfrac{\dfrac{\dfrac{}{A'' \vdash \texttt{Cons} :: \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}], \emptyset}\,(\text{AxK}) \quad , \quad \dfrac{}{A'' \vdash f :: \alpha_1, \emptyset}\,(\text{AxV}) \quad , \quad \dfrac{}{A'' \vdash y :: \alpha_3, \emptyset}\,(\text{AxV})}{A'' \vdash (f\ y) :: \alpha_{15}, \{\alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\,(\text{RApp}) \quad , \quad \dfrac{\dfrac{}{A'' \vdash \texttt{map} :: \beta, \emptyset}\,(\text{AxSK2}) \quad , \quad \dfrac{}{A'' \vdash f :: \alpha_1, \emptyset}\,(\text{AxV})}{A'' \vdash (\texttt{map}\ f) :: \alpha_{12}, \{\beta \doteq \alpha_1 \to \alpha_{12}\}}\,(\text{RApp}) \quad , \quad \dfrac{}{A'' \vdash ys :: \alpha_4, \emptyset}\,(\text{AxV})}{A'' \vdash (\texttt{Cons}\ (f\ y)) :: \alpha_{11}, \{\alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\,(\text{RApp}) \quad , \quad \dfrac{}{A'' \vdash (\texttt{map}\ f\ ys) :: \alpha_{13}, \{\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}\,(\text{RApp})$$

$$\dfrac{}{A'' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \alpha_{14}, \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}$$

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

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

Beispiele: map (4)

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

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

Die Unifikation ergibt

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

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

Beispiele: erneute Betrachtung

```
g x = x : (g (g 'c'))
```

Iteratives Verfahren liefert Fail nach mehreren Iteration.
Hindley-Milner: $A = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$.
Sei $A' = A \cup \{x :: \alpha, \texttt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(\text{AxK}) \quad , \quad \dfrac{}{A' \vdash x :: \alpha, \emptyset}\,(\text{AxV})}{A' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3}\,(\text{RApp}) \quad , \quad \dfrac{}{A' \vdash \texttt{g} :: \beta, \emptyset}\,(\text{AxSK2})}{A' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\beta \doteq \texttt{Char} \to \alpha_7\}}\,(\text{RApp}) \quad , \quad \dfrac{}{A' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(\text{AxK})}{A' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}\,(\text{RApp})$$

$$\dfrac{A' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}{A \vdash_T \texttt{g} :: \sigma(\alpha \to \alpha_2)}\,(\text{MSKRek})$$

wobei σ die Lösung von
$\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$ 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 $g :: \forall \alpha.\alpha \to [\texttt{Int}]$
Hindley-Milner: Sei $A' = A \cup \{x :: \alpha, \texttt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(\text{AxK}) \quad , \quad \dfrac{}{A' \vdash 1 :: \texttt{Int}, \emptyset}\,(\text{AxK})}{A' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}\,(\text{RApp}) \quad , \quad \dfrac{}{A' \vdash \texttt{g} :: \beta, \emptyset}\,(\text{AxSK2})}{A' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\beta \doteq \texttt{Char} \to \alpha_7\}}\,(\text{RApp}) \quad , \quad \dfrac{}{A' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(\text{AxK})}{A' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}\,(\text{RApp})$$

$$\dfrac{A' \vdash \texttt{Cons}\ 1\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}{A \vdash_T \texttt{g} :: \sigma(\alpha \to \alpha_2)}\,(\text{SKRek})$$

wobei σ die Lösung von
$\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$ ist.

Die Unifikation schlägt fehl, da $[\alpha_5] \doteq \texttt{Char}$ 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 :: $\forall a.$ Baum a und
Knoten :: $\forall a.\ a \to$ Baum a $\to$ Baum a $\to$ Baum a

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

Hindley-Milner-Typcheck g :: $a \to a \to$ Baum Bool
Iteratives Verfahren: g :: $a \to b \to$ 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 $S_1, \ldots, S_n$ nicht Hindley-Milner-typisierbar oder Hindley-Milner liefert (vermutet) zu speziellen Typ
- Aktion: Typisiere $S_1, \ldots, S_n$ mit dem iterativen (polymorphen) Typisierungsverfahren
- Wenn Erfolg: $S_1 : \tau_1, \ldots, S_n : \tau_n$,
- Dann: spezifiziere im Programm-File :
- $s_1 :: \tau_1, \ \ldots, \ s_n :: \tau_n$
- 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)

„Vermutlich“: Haskells 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:

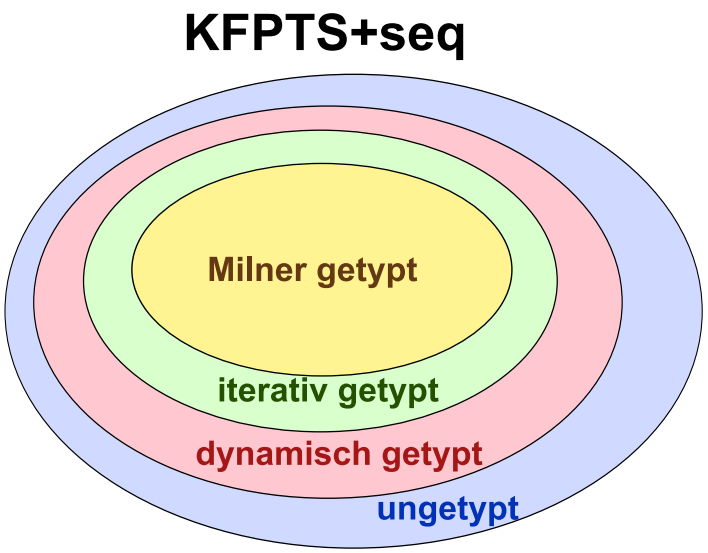
  Vorher: einmal kann allquantifiziert werden
  Nachher: alles wird auf einmal typisiert.

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

Übersicht



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)
```

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 => τ
  wobei C *Typklassenconstraint*, τ 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 τ 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`

Typklassenaxiome

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

1. $Cl\ \tau$ (d.h. $\tau \in Cl$) für Basistypen τ.
   : z.B.: $Cl(\texttt{List Bool})$ ist nicht möglich
2. Implikationen der Form $C \implies Cl(TC\ a_1 \ldots a_n)$
   wobei $a_1, \ldots, a_n$ 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 $Cl\ (TC\ \tau_1 \ldots \tau_n)$ aus $\mathcal{C}$ (mit maximaler Größe); wenn es eine Implikation $C_0 \implies Cl(TC\ a_1 \ldots a_n)$ gibt, dann ersetze das Constraint in $\mathcal{C}$ durch die Menge $\sigma(C_0)$, wobei $\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$ 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:

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

Gilt `Eq ([Int],Bool)`. ?

Start:	$\{$`Eq ([Int], Bool)`$\}$	
	$\{$`Eq [Int], Eq Bool`$\}$	wg. Implikation für Paare
	$\{$`Eq Int, 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 $f\ s_1 \ldots s_n$ der Knoten ist,
   und s_i schon mit a_i markiert ist, dann markiere Knoten mit dem label $f(a_1, \ldots, a_n)$ 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.

Beispiel

Aussagenlogische Auswertung

. . . . .

Typklassenconstraints testen und Baumautomaten

- 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.

Typklassen: Semantik bzgl. Grundtypen

Definition

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

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

$\Longrightarrow$ 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:
$+ :: \{\texttt{Num}\ a\}$ `=>` $a \to a \to a$.

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 $\texttt{Num}\ a$.
Es ergibt sich:
$\lambda x.\lambda y.(x, y, x + y) :: \{\texttt{Num}\ a\}$ `=>` $a \to a \to (a, a, a)$

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 σ 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 σ 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: `(x1,x2) + (y1,y2) = (x1+y1,x2+y2)`
Implikationsaxiom für Paare: $\{\texttt{Num}\ a, \texttt{Num}\ b\} \implies \texttt{Num}\ (a, b)$.

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

$x :: \alpha$.
$+ :: \{\texttt{Num}\ a\}$ => $a \to a \to a$.

Gleichungen: $\alpha \doteq a, a \doteq (\texttt{Int}, \texttt{Int})$
Constraint: $\{\texttt{Num}\ a\}$.
Unifikation ergibt $\sigma = \{a \mapsto (\texttt{Int}, \texttt{Int}), \alpha \mapsto (\texttt{Int}, \texttt{Int})\}$
Constraintmenge: $\{\texttt{Num}\ (\texttt{Int}, \texttt{Int})\}$.
Implikation und Basisconstraints zeigen, dass das Constraint gilt!
Resultat: $f :: (\texttt{Int}, \texttt{Int}) \to (\texttt{Int}, \texttt{Int})$

Typisierung unter Typklassen: Beispiel

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

$x :: \alpha, y :: \beta$
$+ :: \{\texttt{Num}\ a\}$ => $a \to a \to a$.

Gleichungen: $\alpha \doteq (\beta, \beta), a \doteq \alpha)$
Constraint: $\{\texttt{Num}\ a\}$.
Unifikation ergibt $\sigma = \{a \mapsto (\beta, \beta), \alpha \mapsto (\beta, \beta)\}$
Constraintmenge: $\{\texttt{Num}\ (\beta, \beta)\}$.
Implikation ergibt als Constraint: $\{\texttt{Num}\ \beta\}$.
Resultat: $g :: \{\texttt{Num}\ \beta\} \implies (\beta, \beta) \to \beta \to (\beta, \beta)$

Typisierung unter Typklassen: Beispiel

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

Typisierung; ergibt Gleichungen und Bedingungen:
$xs :: \alpha_1$, $0 :: \texttt{Num}\ b_1 \implies b_1$, $1 :: \texttt{Num}\ b_2 \implies b_2$
`genericLength` $:: \alpha_1 \to \alpha_2$.
wegen (`case xs of [] ...`): $\alpha_1 = [\alpha_4]$
`genericLength` $ys :: \alpha_2$
$+ :: \texttt{Num}\ a \implies a \to a \to a$.
$a = b_2$, $a = \alpha_2$, $b_1 = \alpha_2$; $\texttt{Num}\ a$, $\texttt{Num}\ b_1$, $\texttt{Num}\ b_2$.
Ergibt insgesamt: $\texttt{genericLength} :: \texttt{Num}\ a \implies [b] \to a$