

Typisierung

Prof. Dr. David Sabel

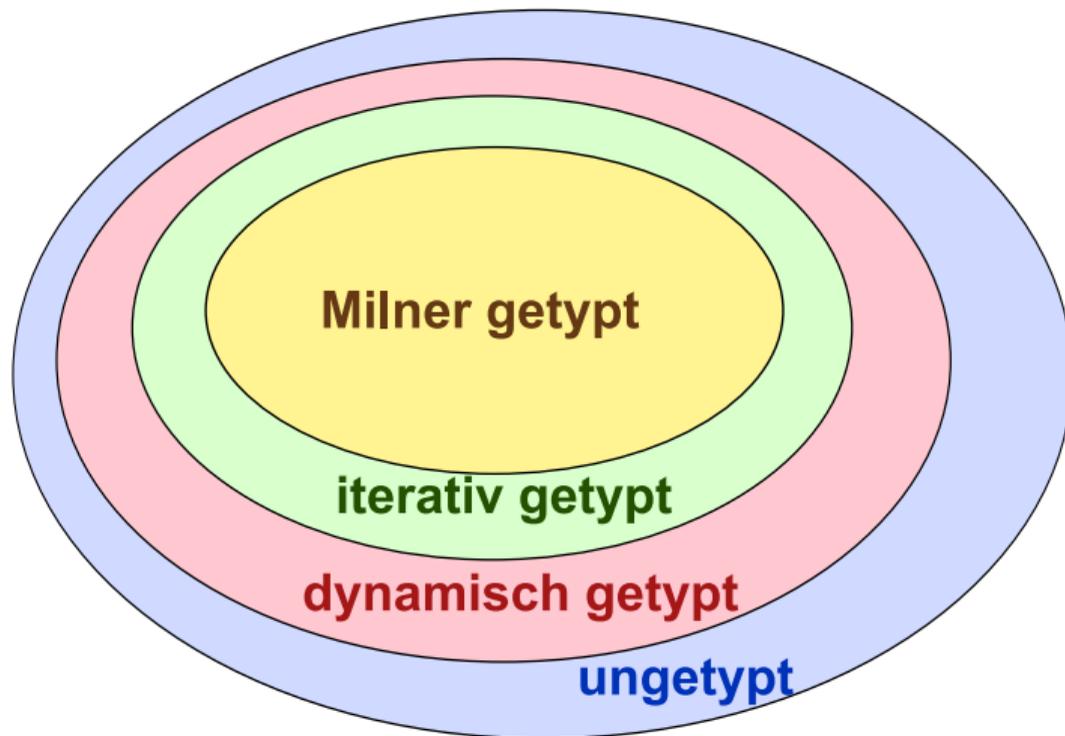
LFE Theoretische Informatik



Ziele des Kapitels

- Warum typisieren?
- Typisierungsverfahren für Haskell bzw. KFPTS+seq für parametrisch polymorphe Typen
- Iteratives Typisierungsverfahren
- Milnersches Typisierungsverfahren

KFPTS+seq



Warum ist ein Typsystem sinnvoll?

- Für ungetypte Programme können **dynamische Typfehler** auftreten
- Fehler zur Laufzeit sind Programmierfehler
- Starkes und statisches Typsystem \implies keine Typfehler zu Laufzeit
- Typen als **Dokumentation**
- Typen bewirken besser strukturierte Programme
- Typen als **Spezifikation** in der Entwurfsphase

Minimalanforderungen:

- Die Typisierung sollte **zur Compilezeit** entschieden werden
- Korrekt getypte Programme erzeugen keine Typfehler zur Laufzeit

Minimalanforderungen:

- Die Typisierung sollte zur **Compilezeit** entschieden werden
- Korrekt getypte Programme erzeugen keine Typfehler zur Laufzeit

Wünschenswerte Eigenschaften:

- Typsystem schränkt wenig oder gar nicht beim Programmieren ein
- Compiler kann selbst Typen berechnen = **Typinferenz**

Motivation (3)

Es gibt Typsysteme, die diese Eigenschaften nicht erfüllen:

- Z.B. **Simply-typed Lambda-Calculus**:
Getypte Sprache ist nicht mehr Turing-mächtig, da dieses Typsystem erzwingt, dass alle Programme **terminieren**
- Erweiterungen in Haskell's Typsystem:
Typisierung / Typinferenz ist unentscheidbar.
U.U. **terminiert der Compiler nicht!**
Folge: Mehr Vorsicht/Anforderungen an den Programmierer.

Naive Definition von „korrekt getypt“:

Ein KFPTS+seq-Programm ist korrekt getypt, wenn es keine dynamischen Typfehler zur Laufzeit erzeugt.

Naive Definition von „korrekt getypt“:

Ein KFPTS+seq-Programm ist korrekt getypt, wenn es keine dynamischen Typfehler zur Laufzeit erzeugt.

Funktioniert **nicht** gut, denn

Die dynamische Typisierung in KFPTS+seq ist **unentscheidbar!**

Unentscheidbarkeit der dynamischen Typisierung

Sei `tmEncode` eine $\text{KFPTS}+\text{seq}$ -Funktion,
die sich wie eine **universelle Turingmaschine** verhält:

- Eingabe: Turingmaschinenbeschreibung und Eingabe für die TM
- Ausgabe: True, falls die Turingmaschine anhält

Beachte: `tmEncode` ist in $\text{KFPTS}+\text{seq}$ definierbar und **nicht dynamisch ungetypt**
(also dynamisch getypt)

Unentscheidbarkeit der dynamischen Typisierung (2)

Für eine TM-Beschreibung b und Eingabe e sei

$$s := \text{if tmEncode } b \ e \\ \quad \text{then case}_{\text{Bool}} \text{ Nil of } \{\text{True} \rightarrow \text{True}; \text{False} \rightarrow \text{False}\} \\ \quad \text{else case}_{\text{Bool}} \text{ Nil of } \{\text{True} \rightarrow \text{True}; \text{False} \rightarrow \text{False}\}$$

Es gilt:

s ist genau dann dynamisch ungetypt, wenn Turingmaschine b auf Eingabe e hält.

Daher: Wenn wir dynamische Typisierung entscheiden könnten, dann auch das Halteproblem

Satz

Die dynamische Typisierung von KFPTS+seq-Programmen ist unentscheidbar.

Syntax von **polymorphen Typen**:

$$\mathbf{T} ::= TV \mid TC \mathbf{T}_1 \dots \mathbf{T}_n \mid \mathbf{T}_1 \rightarrow \mathbf{T}_2$$

wobei TV Typvariable, TC Typkonstruktor

Sprechweisen:

- Ein **Basistyp** ist ein Typ der Form TC , wobei TC ein **nullstelliger** Typkonstruktor ist.
- Ein **Grundtyp** (oder alternativ **monomorpher Typ**) ist ein Typ, der **keine Typvariablen** enthält.

Beispiele:

- `Int`, `Bool` und `Char` sind Basistypen.
- `[Int]` und `Char -> Int` sind keine Basistypen aber Grundtypen.
- `[a]` und `a -> a` sind weder Basistypen noch Grundtypen.

Typen (2)

Wir verwenden für polymorphe Typen die Schreibweise mit All-Quantoren:

- Sei τ ein polymorpher Typ mit Vorkommen der Variablen $\alpha_1, \dots, \alpha_n$
- Dann ist $\forall \alpha_1, \dots, \alpha_n. \tau$ der all-quantifizierte Typ für τ .
- Da Reihenfolge egal, verwenden wir auch $\forall \mathcal{X}. \tau$ wobei \mathcal{X} Menge von Typvariablen

Später:

- Allquantifizierte Typen dürfen kopiert und umbenannt werden,
- Typen ohne Quantor dürfen nicht umbenannt werden!

Typsubstitutionen

Typsubstitution = Abbildung einer endlichen Menge von Typvariablen auf Typen

Schreibweise: $\sigma = \{\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n\}$.

Formal: Erweiterung auf Typen: σ_E : Abbildung von Typen auf Typen

$$\begin{aligned}\sigma_E(TV) &:= \sigma(TV), \text{ falls } \sigma \text{ die Variable } TV \text{ abbildet} \\ \sigma_E(TV) &:= TV, \text{ falls } \sigma \text{ die Variable } TV \text{ nicht abbildet} \\ \sigma_E(TC \ T_1 \ \dots \ T_n) &:= TC \ \sigma_E(T_1) \ \dots \ \sigma_E(T_n) \\ \sigma_E(T_1 \rightarrow T_2) &:= \sigma_E(T_1) \rightarrow \sigma_E(T_2)\end{aligned}$$

Wir unterscheiden im folgenden nicht zwischen σ und der Erweiterung σ_E !

Grundtypen-Semantik für polymorphe Typen:

$$\text{sem}(\tau) := \{\sigma(\tau) \mid \sigma(\tau) \text{ ist Grundtyp}, \sigma \text{ ist Substitution}\}$$

Entspricht der Vorstellung von **schematischen** Typen:

Ein polymorpher Typ ist ein
Schema für eine **Menge von Grundtypen**

Typregeln

Bekannte Regel:

$$\frac{s :: T_1 \rightarrow T_2, \quad t :: T_1}{(s \ t) :: T_2}$$

Problem: Man muss "richtige Instanz raten", z.B.

```
map :: (a -> b)      -> [a] -> [b]
not  :: Bool -> Bool
```

Typisierung von `map not`: Vor Anwendung der Regel muss der Typ von `map` instanziiert werden mit

$$\sigma = \{a \mapsto \text{Bool}, b \mapsto \text{Bool}\}$$

Statt σ zu raten, kann man σ berechnen: **Unifikation**

Definition

- Ein **Unifikationsproblem** auf Typen ist gegeben durch eine Menge E von Gleichungen der Form $\tau_1 \doteq \tau_2$, wobei τ_1 und τ_2 polymorphe Typen sind.
- Eine **Lösung** eines Unifikationsproblem E auf Typen ist eine Substitution σ (bezeichnet als **Unifikator**), so dass $\sigma(\tau_1) = \sigma(\tau_2)$ für alle Gleichungen $\tau_1 \doteq \tau_2$ des Problems.
- Eine **allgemeinste Lösung** (allgemeinster Unifikator, mgu = most general unifier) von E ist ein Unifikator σ , so dass gilt: Für jeden anderen Unifikator ρ von E gibt es eine Substitution γ so dass $\rho(x) = \gamma \circ \sigma(x)$ für alle $x \in FV(E)$.

- Datenstruktur: $E =$ Multimenge von Gleichungen
Multimenge \equiv Menge mit mehrfachem Vorkommen von Elementen
- $E \cup E'$ sei die **disjunkte** Vereinigung von zwei Multimengen
- $E[\tau/\alpha]$ ist definiert als $\{s[\tau/\alpha] \doteq t[\tau/\alpha] \mid (s \doteq t) \in E\}$.

Algorithmus: Wende Schlussregeln (s.u.) solange auf E an, bis

- Fail auftritt, oder
- keine Regel mehr anwendbar ist

Unifikationsalgorithmus: Schlussregeln

Fail-Regeln:

$$\text{FAIL1} \frac{E \cup \{(TC_1 \tau_1 \dots \tau_n) \doteq (TC_2 \tau'_1 \dots \tau'_m)\}}{\text{Fail}}$$

wenn $TC_1 \neq TC_2$

$$\text{FAIL2} \frac{E \cup \{(TC_1 \tau_1 \dots \tau_n) \doteq (\tau'_1 \rightarrow \tau'_2)\}}{\text{Fail}}$$

$$\text{FAIL3} \frac{E \cup \{(\tau'_1 \rightarrow \tau'_2) \doteq (TC_1 \tau_1 \dots \tau_n)\}}{\text{Fail}}$$

Dekomposition:

$$\text{DECOMPOSE1} \frac{E \cup \{TC \tau_1 \dots \tau_n \doteq TC \tau'_1 \dots \tau'_n\}}{E \cup \{\tau_1 \doteq \tau'_1, \dots, \tau_n \doteq \tau'_n\}}$$

$$\text{DECOMPOSE2} \frac{E \cup \{\tau_1 \rightarrow \tau_2 \doteq \tau'_1 \rightarrow \tau'_2\}}{E \cup \{\tau_1 \doteq \tau'_1, \tau_2 \doteq \tau'_2\}}$$

Orientierung, Elimination:

$$\text{ORIENT} \frac{E \cup \{\tau_1 \doteq \alpha\}}{E \cup \{\alpha \doteq \tau_1\}}$$

wenn τ_1 keine Typvariable und α Typvariable

$$\text{ELIM} \frac{E \cup \{\alpha \doteq \alpha\}}{E}$$

wobei α Typvariable

Einsetzung, Occurs-Check:

$$\text{SOLVE } \frac{E \cup \{\alpha \doteq \tau\}}{E[\tau/\alpha] \cup \{\alpha \doteq \tau\}}$$

wenn Typvariable α nicht in τ vorkommt,
aber α kommt in E vor

$$\text{OCCURSCHECK } \frac{E \cup \{\alpha \doteq \tau\}}{\text{Fail}}$$

wenn $\tau \neq \alpha$ und Typvariable α kommt in τ vor

Beispiel 1: $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$:

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

Beispiele

Beispiel 1: $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$:

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

Beispiel 2: $\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$:

$$\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$$

Beispiele

Beispiel 1: $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$:

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

Beispiel 2: $\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$:

$$\text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}$$

Beispiele

Beispiel 1: $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$:

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

Beispiel 2: $\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$:

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}} \\ \text{ORIENT} \frac{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}} \end{array}$$

Beispiele

Beispiel 1: $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$:

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

Beispiel 2: $\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$:

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}} \\ \text{ORIENT} \frac{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}} \\ \text{SOLVE} \frac{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}} \end{array}$$

Beispiele

Beispiel 1: $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$:

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

Beispiel 2: $\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$:

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}} \\ \text{ORIENT} \frac{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}} \\ \text{SOLVE} \frac{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}} \\ \text{SOLVE} \frac{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [\text{Bool}], a \doteq \text{Bool}, c \doteq [\text{Bool}]\}} \end{array}$$

Beispiele

Beispiel 1: $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$:

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

Beispiel 2: $\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$:

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}} \\ \text{ORIENT} \frac{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}} \\ \text{SOLVE} \frac{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}} \\ \text{SOLVE} \frac{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [\text{Bool}], a \doteq \text{Bool}, c \doteq [\text{Bool}]\}} \\ \text{DECOMPOSE1} \frac{\{[d] \doteq [\text{Bool}], a \doteq \text{Bool}, c \doteq [\text{Bool}]\}}{\{d \doteq \text{Bool}, a \doteq \text{Bool}, c \doteq [\text{Bool}]\}} \end{array}$$

Beispiele

Beispiel 1: $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$:

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

Beispiel 2: $\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$:

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}} \\ \text{ORIENT} \frac{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}} \\ \text{SOLVE} \frac{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}} \\ \text{SOLVE} \frac{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [\text{Bool}], a \doteq \text{Bool}, c \doteq [\text{Bool}]\}} \\ \text{DECOMPOSE1} \frac{\{[d] \doteq [\text{Bool}], a \doteq \text{Bool}, c \doteq [\text{Bool}]\}}{\{d \doteq \text{Bool}, a \doteq \text{Bool}, c \doteq [\text{Bool}]\}} \end{array}$$

Der Unifikator ist $\{d \mapsto \text{Bool}, a \mapsto \text{Bool}, c \mapsto [\text{Bool}]\}$.

Beispiele (2)

Beispiel 3: $\{a \dot{=} [b], b \dot{=} [a]\}$

$$\text{OCCURSCHECK} \frac{\text{SOLVE} \frac{\{a \dot{=} [b], b \dot{=} [a]\}}{\{a \dot{=} [[a]], b \dot{=} [a]\}}}{\text{Fail}}$$

Beispiele (2)

Beispiel 3: $\{a \doteq [b], b \doteq [a]\}$

$$\text{OCCURSCHECK} \frac{\text{SOLVE} \frac{\{a \doteq [b], b \doteq [a]\}}{\{a \doteq [[a]], b \doteq [a]\}}}{\text{Fail}}$$

Beispiel 4: $\{a \rightarrow [b] \doteq a \rightarrow c \rightarrow d\}$

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{a \rightarrow [b] \doteq a \rightarrow c \rightarrow d\}}{\{a \doteq a, [b] \doteq c \rightarrow d\}} \\ \text{ELIM} \frac{\{a \doteq a, [b] \doteq c \rightarrow d\}}{\{[b] \doteq c \rightarrow d\}} \\ \text{FAIL2} \frac{\{[b] \doteq c \rightarrow d\}}{\text{Fail}} \end{array}$$

Eigenschaften des Unifikationsalgorithmus

- Der Algorithmus endet mit **Fail** g.d.w. es **keinen Unifikator** für die Eingabe gibt.
- Der Algorithmus endet **erfolgreich** g.d.w. es **einen Unifikator** für die Eingabe gibt. Das Gleichungssystem E ist dann von der Form $\{\alpha_1 \doteq \tau_1, \dots, \alpha_n \doteq \tau_n\}$, wobei α_i paarweise verschiedene Typvariablen sind und kein α_i in irgendeinem τ_j vorkommt. Der Unifikator ist dann $\sigma = \{\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n\}$.
- Liefert der Algorithmus einen Unifikator, dann ist es ein **allgemeinster Unifikator**.
- Man braucht keine alternativen Regelanwendungen auszuprobieren!
Der Algorithmus kann deterministisch implementiert werden.
- Der Algorithmus **terminiert** für jedes Unifikationsproblem auf Typen.
Ausgabe: Fail oder der allgemeinste Unifikator

Eigenschaften des Unifikationsalgorithmus (2)

- Die Typen in der Resultat-Substitution können **exponentiell groß** werden.

Z.B. $\{\alpha_n \doteq \alpha_{n-1} \rightarrow \alpha_{n-1}, \alpha_{n-1} \doteq \alpha_{n-2} \rightarrow \alpha_{n-2}, \dots, \alpha_1 \doteq \alpha_0 \rightarrow \alpha_0\}$

Der Unifikator bildet jedes α_i auf einen Typ ab, der $2^i - 1$ Typpfeile enthält. Z.B.

$$\sigma(\alpha_1) = \alpha_0 \rightarrow \alpha_0,$$

$$\sigma(\alpha_2) = (\alpha_0 \rightarrow \alpha_0) \rightarrow (\alpha_0 \rightarrow \alpha_0),$$

$$\sigma(\alpha_3) = ((\alpha_0 \rightarrow \alpha_0) \rightarrow (\alpha_0 \rightarrow \alpha_0)) \rightarrow ((\alpha_0 \rightarrow \alpha_0) \rightarrow (\alpha_0 \rightarrow \alpha_0))$$

- Der Unifikationsalgorithmus kann so implementiert werden, dass er Zeit $O(n \log n)$ benötigt. Man muss Sharing dazu beachten und eine andere Solve-Regel benutzen. Die Typen in der Resultat-Substitution haben danach Darstellungsgröße $O(n)$.
- Das Unifikationsproblem (d.h. die Frage, ob eine Menge von Typgleichungen unifizierbar ist) ist **P-complete**. D.h. u.a.
 - Man kann im wesentlichen alle PTIME-Probleme als Unifikationsproblem darstellen.
 - Unifikation ist nicht effizient parallelisierbar.

Wir betrachten nun die

polymorphe Typisierung von $\text{KFPTSP}_{+\text{seq}}$ -**Ausdrücken**

Wir verschieben zunächst: Typisierung von **Superkombinatoren**

Wir betrachten nun die

polymorphe Typisierung von $\text{KFPTSP}_{+\text{seq}}$ -Ausdrücken

Wir verschieben zunächst: Typisierung von Superkombinatoren

Nächster Schritt:

Wie müssen die Typisierungsregeln aussehen?

Anwendungsregel mit Unifikation

$$\frac{s :: \tau_1, \quad t :: \tau_2}{(s \ t) :: \sigma(\alpha)}$$

wenn σ allgemeinsten Unifikator für $\tau_1 \doteq \tau_2 \rightarrow \alpha$ ist
und α neue Typvariable ist.

Anwendungsregel mit Unifikation

$$\frac{s :: \tau_1, \quad t :: \tau_2}{(s \ t) :: \sigma(\alpha)}$$

wenn σ allgemeinsten Unifikator für $\tau_1 \doteq \tau_2 \rightarrow \alpha$ ist
und α neue Typvariable ist.

Beispiel:

$$\frac{\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b], \quad \text{not} :: \text{Bool} \rightarrow \text{Bool}}{(\text{map not}) :: \sigma(\alpha)}$$

wenn σ allgemeinsten Unifikator für
 $(a \rightarrow b) \rightarrow [a] \rightarrow [b] \doteq (\text{Bool} \rightarrow \text{Bool}) \rightarrow \alpha$ ist
und α neue Typvariable ist.

Anwendungsregel mit Unifikation

$$\frac{s :: \tau_1, \quad t :: \tau_2}{(s \ t) :: \sigma(\alpha)}$$

wenn σ allgemeinsten Unifikator für $\tau_1 \doteq \tau_2 \rightarrow \alpha$ ist
und α neue Typvariable ist.

Beispiel:

$$\frac{\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b], \quad \text{not} :: \text{Bool} \rightarrow \text{Bool}}{(\text{map not}) :: \sigma(\alpha)}$$

wenn σ allgemeinsten Unifikator für
 $(a \rightarrow b) \rightarrow [a] \rightarrow [b] \doteq (\text{Bool} \rightarrow \text{Bool}) \rightarrow \alpha$ ist
und α neue Typvariable ist.

Unifikation ergibt $\{a \mapsto \text{Bool}, b \mapsto \text{Bool}, \alpha \mapsto [\text{Bool}] \rightarrow [\text{Bool}]\}$

Daher: $\sigma(\alpha) = [\text{Bool}] \rightarrow [\text{Bool}]$

Wie typisiert man eine Abstraktion $\lambda x.s$?

- Typisiere den Rumpf s
- Sei $s :: \tau$
- Dann erhält $\lambda x.s$ einen Funktionstyp $\tau_1 \rightarrow \tau$
- Was hat τ_1 mit τ zu tun?
- τ_1 ist der Typ von x
- Wenn x im Rumpf s vorkommt, brauchen wir τ_1 bei der Berechnung von τ !

Typisierung mit Bindern (2)

Informelle Regel für die Abstraktion:

$$\frac{\text{Typisierung von } s \text{ unter der Annahme " } x \text{ hat Typ } \tau_1 \text{ " ergibt } s :: \tau}{\lambda x. s :: \tau_1 \rightarrow \tau}$$

Woher erhalten wir τ_1 ?

Typisierung mit Bindern (2)

Informelle Regel für die Abstraktion:

$$\frac{\text{Typisierung von } s \text{ unter der Annahme " } x \text{ hat Typ } \tau_1 \text{ " ergibt } s :: \tau}{\lambda x. s :: \tau_1 \rightarrow \tau}$$

Woher erhalten wir τ_1 ?

Nehme allgemeinsten Typ an für x , danach schränke durch die Berechnung von τ den Typ ein.

Beispiel:

- $\lambda x. (x \text{ True})$
- Typisiere $(x \text{ True})$ beginnend mit $x :: \alpha$
- Typisierung muss liefern $\alpha = \text{Bool} \rightarrow \alpha'$
- Typ der Abstraktion $\lambda x. (x \text{ True}) :: (\text{Bool} \rightarrow \alpha') \rightarrow \alpha'$.

Typisierung von Ausdrücken

Erweitertes Regelformat:

$$\Gamma \vdash s :: \tau, E$$

Bedeutung:

Gegeben eine Menge Γ von Typ-Annahmen.

Dann kann für den Ausdruck s der Typ τ und die Typgleichungen E hergeleitet werden.

- In Γ kommen nur Typ-Annahmen für Konstruktoren, Variablen, Superkombinatoren vor.
- In E sammeln wir Gleichungen, sie werden erst später unifiziert.

Typisierung von Ausdrücken (2)

Herleitungsregeln schreiben wir in der Form

$$\frac{\text{Voraussetzung(en)}}{\text{Konsequenz}}$$

$$\frac{\Gamma_1 \vdash s_1 :: \tau_1, E_1 \quad \dots \quad \Gamma_k \vdash s_k :: \tau_k, E_k}{\Gamma \vdash s :: \tau, E}$$

Typisierung von Ausdrücken (2)

Vereinfachung:

Konstruktoranwendungen $(c\ s_1\ \dots\ s_n)$ werden während der Typisierung wie geschachtelte Anwendungen $((((c\ s_1)\ \dots)\ s_n))$ behandelt.

Typisierungsregeln für KFPTS+seq Ausdrücke (1)

Axiom für Variablen:

$$(AxV) \frac{}{\Gamma \cup \{x :: \tau\} \vdash x :: \tau, \emptyset}$$

Typisierungsregeln für KFPTS+seq Ausdrücke (1)

Axiom für Variablen:

$$(AxV) \frac{}{\Gamma \cup \{x :: \tau\} \vdash x :: \tau, \emptyset}$$

Axiom für Konstruktoren:

$$(AxK) \frac{}{\Gamma \cup \{c :: \forall \alpha_1 \dots \alpha_n. \tau\} \vdash c :: \tau[\beta_1/\alpha_1, \dots, \beta_n/\alpha_n], \emptyset}$$

wobei β_i neue Typvariablen sind

- Beachte: Jedesmal wird ein neu umbenannter Typ verwendet!

Typisierungsregeln für KFPTS+seq Ausdrücke (2)

Axiom für Superkombinatoren, deren Typ schon bekannt ist:

$$\text{(AxSK)} \frac{}{\Gamma \cup \{SK :: \forall \alpha_1 \dots \alpha_n. \tau\} \vdash SK :: \tau[\beta_1/\alpha_1, \dots, \beta_n/\alpha_n], \emptyset}$$

wobei β_i neue Typvariablen sind

- Beachte: Jedesmal wird ein neu umbenannter Typ verwendet!

Regel für Anwendungen:

$$(R_{APP}) \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{und} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (s \ t) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha\}}$$

wobei α neue Typvariable

Typisierungsregeln für KFPTS+seq Ausdrücke (3)

Regel für Anwendungen:

$$(R_{APP}) \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{und} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (s \ t) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \dot{=} \tau_2 \rightarrow \alpha\}}$$

wobei α neue Typvariable

Regel für seq:

$$(R_{SEQ}) \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{und} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (\text{seq } s \ t) :: \tau_2, E_1 \cup E_2}$$

Regel für Abstraktionen:

$$(R_{ABS}) \frac{\Gamma \cup \{x :: \alpha\} \vdash s :: \tau, E}{\Gamma \vdash \lambda x.s :: \alpha \rightarrow \tau, E}$$

wobei α eine neue Typvariable

Typisierung eines case: Prinzipien

$$\left(\begin{array}{l} \text{case}_{Typ} s \text{ of } \{ \\ (c_1 \ x_{1,1} \ \dots \ x_{1,ar(c_1)}) \rightarrow t_1; \\ \dots; \\ (c_m \ x_{m,1} \ \dots \ x_{m,ar(c_m)}) \rightarrow t_m \} \end{array} \right)$$

- Die Pattern und der Ausdruck s haben gleichen Typ.
Der Typ muss auch zum Typindex am case passen
(Haskell hat keinen Typindex an case)
- Die Ausdrücke t_1, \dots, t_n haben gleichen Typ,
und dieser Typ ist auch der Typ des ganzen case-Ausdrucks.

Typisierungsregeln für KFPTS+seq Ausdrücke (6)

Regel für case:

$$\begin{array}{l} \Gamma \vdash s :: \tau, E \\ \text{für alle } i = 1, \dots, m: \Gamma \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,ar(c_i)} :: \alpha_{i,ar(c_i)}\} \vdash (c_i x_{i,1} \dots x_{i,ar(c_i)}) :: \tau_i, E_i \\ \text{für alle } i = 1, \dots, m: \Gamma \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,ar(c_i)} :: \alpha_{i,ar(c_i)}\} \vdash t_i :: \tau'_i, E'_i \\ \text{(RCASE)} \hline \Gamma \vdash \left(\text{case}_{Typ} s \text{ of } \left\{ \begin{array}{l} (c_1 x_{1,1} \dots x_{1,ar(c_1)}) \rightarrow t_1; \\ \dots; \\ (c_m x_{m,1} \dots x_{m,ar(c_m)}) \rightarrow t_m \end{array} \right\} \right) :: \alpha, E' \end{array}$$

wobei $E' = E \cup \bigcup_{i=1}^m E_i \cup \bigcup_{i=1}^m E'_i \cup \bigcup_{i=1}^m \{\tau \doteq \tau_i\} \cup \bigcup_{i=1}^m \{\alpha \doteq \tau'_i\}$
und $\alpha_{i,j}, \alpha$ neue Typvariablen sind

Instanz der Case-Regel für Bool

$$\text{(RCASE)} \frac{\Gamma \vdash s :: \tau, E \quad \Gamma \vdash \text{True} :: \tau_1, E_1 \quad \Gamma \vdash \text{False} :: \tau_2, E_2 \quad \Gamma \vdash t_1 :: \tau'_1, E'_1 \quad \Gamma \vdash t_2 :: \tau'_2, E'_2}{\Gamma \vdash (\text{case}_{\text{Bool}} s \text{ of } \{\text{True} \rightarrow t_1; \text{False} \rightarrow t_2\}) :: \alpha, E'}$$

wobei $E' = E \cup E_1 \cup E_2 \cup E'_1 \cup E'_2 \cup \{\tau \doteq \tau_1, \tau \doteq \tau_2\} \cup \{\alpha \doteq \tau'_1, \alpha \doteq \tau'_2\}$
und $\alpha_{i,j}, \alpha$ neue Typvariablen sind

Instanz der Case-Regel für Listen

$$\begin{array}{l} \Gamma \vdash s :: \tau, E \\ \Gamma \vdash \text{Nil} :: \tau_1, E_1 \\ \Gamma \cup \{x_1 :: \alpha_1, x_2 :: \alpha_2\} \vdash \text{Cons } x_1 \ x_2 :: \tau_2, E_2 \\ \Gamma \vdash t_1 :: \tau'_1, E'_1 \\ \Gamma \cup \{x_1 :: \alpha_1, x_2 :: \alpha_2\} \vdash t_2 :: \tau'_2, E'_2 \end{array}$$

$$\text{(RCASE)} \frac{}{\Gamma \vdash (\text{case}_{\text{List}} s \text{ of } \{\text{Nil} \rightarrow t_1; (\text{Cons } x_1 \ x_2) \rightarrow t_2\}) :: \alpha, E'}$$

wobei $E' = E \cup E_1 \cup E_2 \cup E'_1 \cup E'_2 \cup \{\tau \doteq \tau_1, \tau \doteq \tau_2\} \cup \{\alpha \doteq \tau'_1, \alpha \doteq \tau'_2\}$
und $\alpha_{i,j}, \alpha$ neue Typvariablen sind

Typisierungsalgorithmus für KFPTS+seq-Ausdrücke

Sei s ein geschlossener KFPTS+seq-Ausdruck, wobei die Typen für alle in s benutzten Superkombinatoren und Konstruktoren bekannt sind.

- 1 Starte mit Anfangsannahme Γ , die Typen für die Konstruktoren und die Superkombinatoren enthält.
- 2 Leite $\Gamma \vdash s :: \tau, E$ mit den Typisierungsregeln her.
- 3 Löse E mit Unifikation.
- 4 Wenn die Unifikation mit Fail endet, ist s nicht typisierbar;
Andernfalls: Sei σ ein allgemeinsten Unifikator von E , dann gilt $s :: \sigma(\tau)$.

Zusätzliche Regel, zum zwischendrin Unifizieren:

Typberechnung:

$$\text{(RU}_{\text{NIF}}) \frac{\Gamma \vdash s :: \tau, E}{\Gamma \vdash s :: \sigma(\tau), E_{\sigma}}$$

wobei E_{σ} das gelöste Gleichungssystem zu E ist
und σ der ablesbare Unifikator ist

Definition

Ein $\text{KFPTS}_{+\text{seq}}$ Ausdruck s ist **wohl-getypt**, wenn er sich mit obigem Verfahren typisieren lässt.

(Typisierung von Superkombinatoren kommt noch)

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

$$\text{(RAPP)} \frac{\Gamma_0 \vdash (\text{Cons True}) :: \tau_1, E_1, \quad \Gamma_0 \vdash \text{Nil} :: \tau_2, E_2}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha_4\}}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a.a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a.[a], \text{True} :: \text{Bool}\}$

$$\text{(RAPP)} \frac{\Gamma_0 \vdash (\text{Cons True}) :: \tau_1, E_1, \quad \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset}}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, E_1 \cup \emptyset \cup \{\tau_1 \doteq [\alpha_3] \rightarrow \alpha_4\}}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a.a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a.[a], \text{True} :: \text{Bool}\}$

$$\begin{array}{c} \text{(RAPP)} \frac{\Gamma_0 \vdash \text{Cons} :: \tau_3, E_3, \Gamma_0 \vdash \text{True} :: \tau_4, E_4}{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_2\} \cup E_3 \cup E_4}, \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_2\} \cup E_3 \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}} \end{array}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

$$\begin{array}{c} \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset, \Gamma_0 \vdash \text{True} :: \tau_4, E_4} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \tau_4 \rightarrow \alpha_2\} \cup E_4, \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \tau_4 \rightarrow \alpha_2\} \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}} \end{array}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

$$\begin{array}{c} \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \quad \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{True} :: \text{Bool}, \emptyset} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \quad \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\} \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}} \end{array}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

$$\begin{array}{c} \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \quad \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{True} :: \text{Bool}, \emptyset} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \quad \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}} \end{array}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

$$\begin{array}{c} \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \quad \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{True} :: \text{Bool}, \emptyset} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \quad \text{(AxK)} \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\ \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}} \end{array}$$

Löse $\{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}$ mit Unifikation

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

$$\frac{\frac{\text{(AxK)} \overline{\Gamma_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \text{(AxK)} \overline{\Gamma_0 \vdash \text{True} :: \text{Bool}, \emptyset}}{\text{(RAPP)} \overline{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \text{(AxK)} \overline{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset}}{\text{(RAPP)} \overline{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}}$$

Löse $\{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}$ mit Unifikation

Ergibt: $\sigma = \{\alpha_1 \mapsto \text{Bool}, \alpha_2 \mapsto ([\text{Bool}] \rightarrow [\text{Bool}]), \alpha_3 \mapsto \text{Bool}, \alpha_4 \mapsto [\text{Bool}]\}$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $\Gamma_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

$$\frac{\frac{\text{(AxK)} \overline{\Gamma_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \text{(AxK)} \overline{\Gamma_0 \vdash \text{True} :: \text{Bool}, \emptyset}}{\text{(RAPP)} \overline{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \text{(AxK)} \overline{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset}}{\text{(RAPP)} \overline{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}}$$

Löse $\{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}$ mit Unifikation

Ergibt: $\sigma = \{\alpha_1 \mapsto \text{Bool}, \alpha_2 \mapsto ([\text{Bool}] \rightarrow [\text{Bool}]), \alpha_3 \mapsto \text{Bool}, \alpha_4 \mapsto [\text{Bool}]\}$

Daher $(\text{Cons True Nil}) :: \sigma(\alpha_4) = [\text{Bool}]$

Beispiele: Typisierung von $\lambda x.x$

Typisierung von $\lambda x.x$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\text{(R}_{\text{ABS}}\text{)} \frac{\Gamma_0 \cup \{x :: \alpha\} \vdash x :: \tau, E}{\Gamma_0 \vdash (\lambda x.x) :: \alpha \rightarrow \tau, E}$$

Beispiele: Typisierung von $\lambda x.x$

Typisierung von $\lambda x.x$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\begin{array}{c} \text{(AxV)} \frac{}{\Gamma_0 \cup \{x :: \alpha\} \vdash x :: \alpha, \emptyset} \\ \text{(RAbs)} \frac{}{\Gamma_0 \vdash (\lambda x.x) :: \alpha \rightarrow \alpha, \emptyset} \end{array}$$

Beispiele: Typisierung von $\lambda x.x$

Typisierung von $\lambda x.x$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\begin{array}{c} \text{(AxV)} \frac{}{\Gamma_0 \cup \{x :: \alpha\} \vdash x :: \alpha, \emptyset} \\ \text{(RAbs)} \frac{}{\Gamma_0 \vdash (\lambda x.x) :: \alpha \rightarrow \alpha, \emptyset} \end{array}$$

Nichts zu unifizieren, daher $(\lambda x.x) :: \alpha \rightarrow \alpha$

Beispiele: Typisierung von Ω

Typisierung von $(\lambda x.(x x)) (\lambda y.(y y))$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\text{(RAPP)} \frac{\emptyset \vdash (\lambda x.(x x)) :: \tau_1, E_1, \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha_1\}}$$

Beispiele: Typisierung von Ω

Typisierung von $(\lambda x.(x x)) (\lambda y.(y y))$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\begin{array}{c} \text{(RABS)} \frac{\{x :: \alpha_2\} \vdash (x x) :: \tau_1, E_1}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \tau_1, E_1}, \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2 \\ \text{(RAPP)} \frac{}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha_1\}} \end{array}$$

Beispiele: Typisierung von Ω

Typisierung von $(\lambda x.(x x)) (\lambda y.(y y))$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\begin{array}{c} \text{(RAPP)} \frac{\{x :: \alpha_2\} \vdash x :: \tau_3, E_3, \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,}{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4} \\ \text{(RABS)} \frac{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4, \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4 \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}} \end{array}$$

Beispiele: Typisierung von Ω

Typisierung von $(\lambda x.(x x)) (\lambda y.(y y))$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\begin{array}{c} \text{(AXV)} \frac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset, \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,} \\ \text{(RAPP)} \frac{}{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\alpha_2 \dot{=} \tau_4 \rightarrow \alpha_3\} \cup E_4} \\ \text{(RABS)} \frac{}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\alpha_2 \dot{=} \tau_4 \rightarrow \alpha_3\} \cup E_4, \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2} \\ \text{(RAPP)} \frac{}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \dot{=} \tau_4 \rightarrow \alpha_3\} \cup E_4 \cup E_2 \cup \{\alpha_3 \dot{=} \tau_2 \rightarrow \alpha_1\}} \end{array}$$

Beispiele: Typisierung von Ω

Typisierung von $(\lambda x.(x x)) (\lambda y.(y y))$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\begin{array}{c} \text{(AxV)} \frac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}, \quad \text{(AxV)} \frac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}, \\ \text{(RAPP)} \frac{}{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\alpha_2 \doteq \alpha_2 \rightarrow \alpha_3\}} \\ \text{(RABS)} \frac{}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\alpha_2 \doteq \alpha_2 \rightarrow \alpha_3\}}, \quad \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2 \\ \text{(RAPP)} \frac{}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \doteq \alpha_2 \rightarrow \alpha_3\} \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}} \end{array}$$

Beispiele: Typisierung von Ω

Typisierung von $(\lambda x.(x x)) (\lambda y.(y y))$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\begin{array}{c} \text{(AxV)} \frac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}, \quad \text{(AxV)} \frac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}, \\ \text{(RAPP)} \frac{}{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\alpha_2 \doteq \alpha_2 \rightarrow \alpha_3\}} \quad \dots \\ \text{(RABS)} \frac{}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\alpha_2 \doteq \alpha_2 \rightarrow \alpha_3\}} \quad , \quad \frac{}{\emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2} \\ \text{(RAPP)} \frac{}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \doteq \alpha_2 \rightarrow \alpha_3\} \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}} \end{array}$$

Beispiele: Typisierung von Ω

Typisierung von $(\lambda x.(x x)) (\lambda y.(y y))$

Starte mit: Anfangsannahme: $\Gamma_0 = \emptyset$

$$\begin{array}{c} \text{(AxV)} \frac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}, \quad \text{(AxV)} \frac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}, \\ \text{(RAPP)} \frac{}{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\alpha_2 \dot{=} \alpha_2 \rightarrow \alpha_3\}} \\ \text{(RABS)} \frac{}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\alpha_2 \dot{=} \alpha_2 \rightarrow \alpha_3\}} \quad \dots \\ \text{(RAPP)} \frac{}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \dot{=} \alpha_2 \rightarrow \alpha_3\} \cup E_2 \cup \{\alpha_3 \dot{=} \tau_2 \rightarrow \alpha_1\}} \end{array}$$

Man sieht schon:

Die Unifikation schlägt fehl, wegen: $\alpha_2 \dot{=} \alpha_2 \rightarrow \alpha_3$

Daher: $(\lambda x.(x x)) (\lambda y.(y y))$ ist **nicht typisierbar!**

Beachte: $(\lambda x.(x x)) (\lambda y.(y y))$ ist **nicht dynamisch ungetypt** aber nicht wohl-getypt

Beispiele: Typisierung eines Ausdrucks mit SKs (1)

Annahme: `map` und `length` sind bereits typisierte Superkombinatoren.

Wir typisieren:

$$t := \lambda xs. \text{case}_{\text{List}} xs \text{ of } \{ \text{Nil} \rightarrow \text{Nil}; (\text{Cons } y \ ys) \rightarrow \text{map length } ys \}$$

Als Anfangsannahme benutzen wir:

$$\Gamma_0 = \{ \begin{array}{l} \text{map} :: \forall a, b. (a \rightarrow b) \rightarrow [a] \rightarrow [b], \\ \text{length} :: \forall a. [a] \rightarrow \text{Int}, \\ \text{Nil} :: \forall a. [a] \\ \text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a] \\ \end{array} \}$$

Beispiele: Typisierung eines Ausdrucks mit SKs (5)

Beschriftung unten:

$$\begin{aligned} B_1 = & \Gamma_0 \vdash t :: \alpha_1 \rightarrow \alpha_{13}, \\ & \{ \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7, \\ & (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}, \\ & \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \} \end{aligned}$$

Beispiele: Typisierung eines Ausdrucks mit SKs (5)

Beschriftung unten:

$$\begin{aligned} B_1 = \quad & \Gamma_0 \vdash t :: \alpha_1 \rightarrow \alpha_{13}, \\ & \{ \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7, \\ & (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}, \\ & \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \} \end{aligned}$$

Löse mit Unifikation:

$$\begin{aligned} & \{ \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7, \\ & (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}, \\ & \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12} \} \end{aligned}$$

Beispiele: Typisierung eines Ausdrucks mit SKs (5)

Beschriftung unten:

$$B_1 = \Gamma_0 \vdash t :: \alpha_1 \rightarrow \alpha_{13}, \\ \{\alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7, \\ (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}, \\ \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}$$

Löse mit Unifikation:

$$\{\alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7, \\ (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}, \\ \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}\}$$

Ergibt:

$$\sigma = \{\alpha_1 \mapsto [[\alpha_{10}]], \alpha_2 \mapsto [\alpha_{10}], \alpha_3 \mapsto [\alpha_{10}], \alpha_4 \mapsto [[\alpha_{10}]], \alpha_5 \mapsto [\alpha_{10}], \\ \alpha_6 \mapsto [[\alpha_{10}]] \rightarrow [[\alpha_{10}]], \alpha_7 \mapsto [[\alpha_{10}]], \alpha_8 \mapsto [\alpha_{10}], \alpha_9 \mapsto \mathbf{Int}, \\ \alpha_{11} \mapsto [[\alpha_{10}]] \rightarrow [\mathbf{Int}], \alpha_{12} \mapsto [\mathbf{Int}], \alpha_{13} \mapsto [\mathbf{Int}], \alpha_{14} \mapsto \mathbf{Int}\}$$

Damit erhält man $t :: \sigma(\alpha_1 \rightarrow \alpha_{13}) = [[\alpha_{10}]] \rightarrow [\mathbf{Int}]$.

Bsp.: Typisierung von Lambda-geb. Variablen (1)

Die Funktion `const` ist definiert als

```
const :: a -> b -> a
```

```
const x y = x
```

Typisierung von `$\lambda x.\text{const } (x \text{ True}) (x \text{ 'A'})$`

Anfangsannahme:

$$\Gamma_0 = \{\text{const} :: \forall a, b. a \rightarrow b \rightarrow a, \text{True} :: \text{Bool}, \text{'A'} :: \text{Char}\}.$$

Bsp.: Typisierung von Lambda-geb. Variablen (3)

In Haskell:

```
Main> \x -> const (x True) (x 'A')
```

```
<interactive>:1:23:
```

```
Couldn't match expected type `Char' against inferred type `Bool'
```

```
  Expected type: Char -> b
```

```
  Inferred type: Bool -> a
```

```
In the second argument of `const', namely `(x 'A)'
```

```
In the expression: const (x True) (x 'A')
```

Bsp.: Typisierung von Lambda-geb. Variablen (3)

In Haskell:

```
Main> \x -> const (x True) (x 'A')
```

```
<interactive>:1:23:
```

```
Couldn't match expected type `Char' against inferred type `Bool'
```

```
  Expected type: Char -> b
```

```
  Inferred type: Bool -> a
```

```
In the second argument of `const', namely `(x 'A)'
```

```
In the expression: const (x True) (x 'A')
```

- Beispiel verdeutlicht: **Lambda-gebundene Variablen** sind **monomorph** getypt!
- Das gleiche gilt für case-Pattern gebundene Variablen

Bsp.: Typisierung von Lambda-geb. Variablen (3)

In Haskell:

```
Main> \x -> const (x True) (x 'A')
```

```
<interactive>:1:23:
```

```
Couldn't match expected type `Char' against inferred type `Bool'
```

```
Expected type: Char -> b
```

```
Inferred type: Bool -> a
```

```
In the second argument of `const', namely `(x 'A)'
```

```
In the expression: const (x True) (x 'A')
```

- Beispiel verdeutlicht: **Lambda-gebundene Variablen** sind **monomorph** getypt!
- Das gleiche gilt für case-Pattern gebundene Variablen
- Daher spricht man auch von **let-Polymorphismus**, da nur let-gebundene Variablen polymorph sind.
- KFPTS+seq hat kein let, aber **Superkombinatoren**, die wie (ein eingeschränktes rekursives) let wirken

Definition (direkt rekursiv, rekursiv, verschränkt rekursiv)

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

$$SK_i \preceq SK_j$$

g.d.w. 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: $\Gamma \vdash_T SK :: \tau$, bedeutet:
unter Annahme Γ kann man SK mit Typ τ typisieren

Typisierung von nicht-rekursiven Superkombinatoren

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

Typisierungsregel für (geschlossene) nicht-rekursive SK:

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

wenn σ 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 $\sigma(\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau)$

Beispiel: Typisierung von $(.)$

$(.) \text{ f g x} = \text{f (g x)}$

Γ_0 ist leer, da keine Konstruktoren oder SK vorkommen.

$$\frac{\frac{\frac{\frac{\text{(AxV)} \overline{\Gamma_1 \vdash f :: \alpha_1, \emptyset}}{\text{(RAPP)} \overline{\Gamma_1 \vdash (f (g x)) :: \alpha_4, \{\alpha_2 \dot{=} \alpha_3 \rightarrow \alpha_5, \alpha_1 = \alpha_5 \rightarrow \alpha_4\}}}{\text{(RSK1)} \overline{\emptyset \vdash_T (. :: \forall \mathcal{X}. \sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4)}}}{\text{(AxV)} \overline{\Gamma_1 \vdash f :: \alpha_1, \emptyset}, \frac{\frac{\text{(AxV)} \overline{\Gamma_1 \vdash g :: \alpha_2, \emptyset}, \frac{\text{(AxV)} \overline{\Gamma_1 \vdash x :: \alpha_3, \emptyset}}{\text{(RAPP)} \overline{\Gamma_1 \vdash (g x) :: \alpha_5, \{\alpha_2 \dot{=} \alpha_3 \rightarrow \alpha_5\}}}}{\text{(RAPP)} \overline{\Gamma_1 \vdash (f (g x)) :: \alpha_4, \{\alpha_2 \dot{=} \alpha_3 \rightarrow \alpha_5, \alpha_1 = \alpha_5 \rightarrow \alpha_4\}}}}{\text{(RSK1)} \overline{\emptyset \vdash_T (. :: \forall \mathcal{X}. \sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4)}}}$$

wobei $\Gamma_1 = \{f :: \alpha_1, g :: \alpha_2, x :: \alpha_3\}$

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:

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

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 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 Typ weiter
- Stoppe, wenn **neuer Typ = alter Typ**
- Dann hat man eine **konsistente Typannahme** gefunden;
Vermutung: auch eine ausreichend allgemeine (allgemeinste?)

Allgemeinster Typ: Typ T so dass $\text{sem}(T) = \{\text{alle Grundtypen}\}$.

Das liefert der Typ α (bzw. quantifiziert $\forall \alpha. \alpha$)

Regel zur Berechnung neuer Annahmen:

$$(SK_{REK}) \frac{\Gamma \cup \{x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SK :: \sigma(\alpha_1 \rightarrow \dots \alpha_n \rightarrow \tau)}$$

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

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

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:

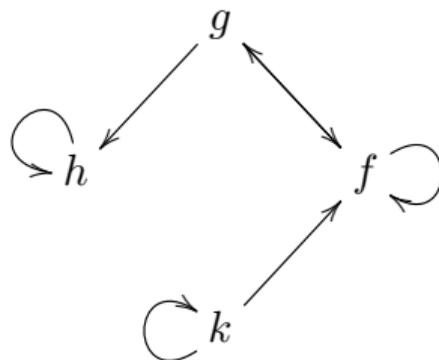
$f\ x\ y = \text{if } x \leq 1 \text{ then } y \text{ else } f\ (x-y)\ (y + g\ x)$

$g\ x = \text{if } x == 0 \text{ then } (f\ 1\ x) + (h\ 2) \text{ else } 10$

$h\ x = \text{if } x == 1 \text{ then } 0 \text{ else } h\ (x-1)$

$k\ x\ y = \text{if } x == 1 \text{ then } y \text{ else } k\ (x-1)\ (y+(f\ x\ y))$

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



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

Iterativer Typisierungsalgorithmus

Eingabe: Verschränkt rekursive Superkombinatoren SK_1, \dots, SK_m (kleinere SKs schon typisiert)

- 1 Anfangsannahme Γ enthält Typen der Konstruktoren und der bereits bekannten SKs

Iterativer Typisierungsalgorithmus

Eingabe: Verschränkt rekursive Superkombinatoren SK_1, \dots, SK_m (kleinere SKs schon typisiert)

- 1 Anfangsannahme Γ enthält Typen der Konstruktoren und der bereits bekannten SKs
- 2 $\Gamma_0 := \Gamma \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$ und $j = 0$.

Iterativer Typisierungsalgorithmus

Eingabe: Verschränkt rekursive Superkombinatoren SK_1, \dots, SK_m (kleinere SKs schon typisiert)

- 1 Anfangsannahme Γ enthält Typen der Konstruktoren und der bereits bekannten SKs
- 2 $\Gamma_0 := \Gamma \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$ und $j = 0$.
- 3 Verwende für jeden Superkombinator SK_i (mit $i = 1, \dots, m$) die Regel (SKREK) und Annahme Γ_j , um SK_i zu typisieren.

Iterativer Typisierungsalgorithmus

Eingabe: Verschränkt rekursive Superkombinatoren SK_1, \dots, SK_m (kleinere SKs schon typisiert)

- 1 Anfangsannahme Γ enthält Typen der Konstruktoren und der bereits bekannten SKs
- 2 $\Gamma_0 := \Gamma \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$ und $j = 0$.
- 3 Verwende für jeden Superkombinator SK_i (mit $i = 1, \dots, m$) die Regel (SKREK) und Annahme Γ_j , um SK_i zu typisieren.
- 4 Wenn die m Typisierungen erfolgreich, d.h. für alle $i: \Gamma_j \vdash_T SK_i :: \tau_i$
Dann allquantifiziere: $SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m$
Setze $\Gamma_{j+1} := \Gamma \cup \{SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m\}$

Iterativer Typisierungsalgorithmus

Eingabe: Verschränkt rekursive Superkombinatoren SK_1, \dots, SK_m (kleinere SKs schon typisiert)

- 1 Anfangsannahme Γ enthält Typen der Konstruktoren und der bereits bekannten SKs
- 2 $\Gamma_0 := \Gamma \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$ und $j = 0$.
- 3 Verwende für jeden Superkombinator SK_i (mit $i = 1, \dots, m$) die Regel (SKREK) und Annahme Γ_j , um SK_i zu typisieren.
- 4 Wenn die m Typisierungen erfolgreich, d.h. für alle $i: \Gamma_j \vdash_T SK_i :: \tau_i$
Dann allquantifiziere: $SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m$
Setze $\Gamma_{j+1} := \Gamma \cup \{SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m\}$
- 5 Wenn $\Gamma_j \neq \Gamma_{j+1}$, dann gehe mit $j := j + 1$ zu Schritt (3).
Anderenfalls, d.h. wenn $\Gamma_j = \Gamma_{j+1}$, war Γ_j **konsistent**.

Iteratives Typisierungsverfahren: Der Algorithmus

Iterativer Typisierungsalgorithmus

Eingabe: Verschränkt rekursive Superkombinatoren SK_1, \dots, SK_m (kleinere SKs schon typisiert)

- 1 Anfangsannahme Γ enthält Typen der Konstruktoren und der bereits bekannten SKs
- 2 $\Gamma_0 := \Gamma \cup \{SK_1 :: \forall \alpha_1. \alpha_1, \dots, SK_m :: \forall \alpha_m. \alpha_m\}$ und $j = 0$.
- 3 Verwende für jeden Superkombinator SK_i (mit $i = 1, \dots, m$) die Regel (SKREK) und Annahme Γ_j , um SK_i zu typisieren.
- 4 Wenn die m Typisierungen erfolgreich, d.h. für alle $i: \Gamma_j \vdash_T SK_i :: \tau_i$
Dann allquantifiziere: $SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m$
Setze $\Gamma_{j+1} := \Gamma \cup \{SK_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SK_m :: \forall \mathcal{X}_m. \tau_m\}$
- 5 Wenn $\Gamma_j \neq \Gamma_{j+1}$, dann gehe mit $j := j + 1$ zu Schritt (3).
Anderenfalls, d.h. wenn $\Gamma_j = \Gamma_{j+1}$, war Γ_j **konsistent**.

Ausgabe: Allquantifizierte polymorphen Typen der SK_i aus der konsistenten Annahme.

Sollte irgendwann ein Fail in der Unifikation auftreten, 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.
- 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:

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

1.Iteration: $\Gamma_0 = \Gamma \cup \{\text{length} :: \forall \alpha.\alpha\}$

$$\begin{array}{l} (a) \quad \Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \tau_1, E_1 \\ (b) \quad \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \text{Nil} :: \tau_2, E_2 \\ (c) \quad \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (y : ys) :: \tau_3, E_3 \\ (d) \quad \Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \tau_4, E_4 \\ (e) \quad \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (1 + \text{length } ys) :: \tau_5, E_5 \\ \hline \text{(RCASE)} \quad \Gamma_0 \cup \{xs :: \alpha_1\} \vdash (\text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow 0; (y : ys) \rightarrow 1 + \text{length } xs\}) :: \alpha_3, \\ \quad E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \dot{=} \tau_2, \tau_1 \dot{=} \tau_3, \alpha_3 \dot{=} \tau_4, \alpha_3 \dot{=} \tau_5\} \\ \hline \text{(SKREK)} \quad \Gamma_0 \vdash_T \text{length} :: \sigma(\alpha_1 \rightarrow \alpha_3) \end{array}$$

wobei σ 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): $(\text{AxV}) \frac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset}$
 D.h. $\tau_1 = \alpha_1$ und $E_1 = \emptyset$

(b): $(\text{AxK}) \frac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash \text{Nil} :: [\alpha_6], \emptyset}$
 D.h. $\tau_2 = [\alpha_6]$ und $E_2 = \emptyset$

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

wobei $\Gamma_0 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

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

Beispiele: length (3)

(d)
$$\frac{(\text{AxK})}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \text{Int}, \emptyset}$$

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

(e)
$$\frac{\frac{(\text{AxK})}{\Gamma'_0 \vdash (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \emptyset}, \frac{(\text{AxK})}{\Gamma'_0 \vdash 1 :: \text{Int}, \emptyset}}{\frac{(\text{RApP})}{\Gamma'_0 \vdash ((+) 1) :: \alpha_{11}, \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \doteq \text{Int} \rightarrow \alpha_{11}\}}, \frac{(\text{AxSK})}{\Gamma'_0 \vdash \text{length} :: \alpha_{13}, \emptyset}, \frac{(\text{AxV})}{\Gamma'_0 \vdash (ys) :: \alpha_5, \emptyset}}{\frac{(\text{RApP})}{\Gamma'_0 \vdash (\text{length } ys) :: \alpha_{12}, \{\alpha_{13} \doteq \alpha_5 \rightarrow \alpha_{12}\}}}, \frac{(\text{RApP})}{\Gamma'_0 \vdash (1 + \text{length } ys) :: \alpha_{10}, \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \doteq \text{Int} \rightarrow \alpha_{11}, \alpha_{13} \doteq \alpha_5 \rightarrow \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \rightarrow \alpha_{10}\}}$$

 wobei $\Gamma_0 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

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

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

Beispiele: length (4)

Zusammengefasst: $\Gamma_0 \vdash_T \text{length} :: \sigma(\alpha_1 \rightarrow \alpha_3)$

wobei σ Lösung von

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

Die Unifikation ergibt als Unifikator

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

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

$$\Gamma_1 = \Gamma \cup \{ \text{length} :: \forall \alpha. [\alpha] \rightarrow \text{Int} \}$$

Da $\Gamma_0 \neq \Gamma_1$ muss man mit Γ_1 erneut iterieren.

2.Iteration: Ergibt den gleichen Typ, daher war Γ_1 konsistent.

Iteratives Verfahren ist allgemeiner als Haskell

Beispiel

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

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

$\Gamma_0 = \Gamma \cup \{g :: \forall \alpha. \alpha\}$ (und $\Gamma'_0 = \Gamma_0 \cup \{x :: \alpha_1\}$):

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{\Gamma'_0 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}{(\text{AxK})}, \frac{\Gamma'_0 \vdash 1 :: \text{Int}, \emptyset}{(\text{AxK})}}{\Gamma'_0 \vdash (\text{Cons } 1) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \text{Int} \rightarrow \alpha_3}{(\text{RAPP})}, \frac{\frac{\frac{\frac{\frac{\Gamma'_0 \vdash g :: \alpha_6, \emptyset}{(\text{AxSK})}, \frac{\Gamma'_0 \vdash (g\ 'c') :: \alpha_7, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7\}}{(\text{RAPP})}}{\Gamma'_0 \vdash (g\ (g\ 'c')) :: \alpha_4, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4\}}{(\text{RAPP})}}{\Gamma'_0 \vdash \text{Cons } 1\ (g\ (g\ 'c')) :: \alpha_2, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \text{Int} \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}}{(\text{SKREK})}}{\Gamma_0 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_1 \rightarrow [\text{Int}]}$$

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 von $\{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \text{Int} \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}$ ist.

D.h. $\Gamma_1 = \Gamma \cup \{g :: \forall \alpha. \alpha \rightarrow [\text{Int}]\}$.

Nächste Iteration zeigt: Γ_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)

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

- $\Gamma = \{\text{Cons} :: \forall a.a \rightarrow [a] \rightarrow [a], 'c' :: \text{Char}\}$.
- $\Gamma_0 = \Gamma \cup \{g :: \forall \alpha.\alpha\}$

$$\frac{\frac{\frac{\frac{\frac{\frac{\Gamma'_0 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}{(\text{AxK})}, \frac{\Gamma'_0 \vdash x :: \alpha_1, \emptyset}{(\text{AxV})}}{\Gamma'_0 \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3}{(\text{RAPP})}, \frac{\frac{\frac{\Gamma'_0 \vdash g :: \alpha_6, \emptyset}{(\text{AxSK})}, \frac{\Gamma'_0 \vdash (g\ 'c') :: \alpha_7, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7\}}{(\text{RAPP})}}{\Gamma'_0 \vdash (g\ (g\ 'c')) :: \alpha_4, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4\}}{(\text{RAPP})}}{\Gamma'_0 \vdash \text{Cons } x\ (g\ (g\ 'c')) :: \alpha_2, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}}{(\text{SKREK})}}{\Gamma_0 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_5 \rightarrow [\alpha_5]}$$

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 $\{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}$ ist.

D.h. $\Gamma_1 = \Gamma \cup \{g :: \forall \alpha.\alpha \rightarrow [\alpha]\}$.

Bsp.: Mehrere Iterationen sind nötig (2)

Da $\Gamma_0 \neq \Gamma_1$ muss eine weitere Iteration durchgeführt werden.

Sei $\Gamma'_1 = \Gamma_1 \cup \{x :: \alpha_1\}$:

$$\begin{array}{c}
 \frac{\frac{\frac{\frac{\frac{\frac{\Gamma'_1 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}{(\text{AxK})}, \frac{\Gamma'_1 \vdash x :: \alpha_1, \emptyset}{(\text{AxV})}}{\Gamma'_1 \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3}{(\text{RAPP})}, \frac{\frac{\frac{\frac{\frac{\Gamma'_1 \vdash g :: \alpha_6 \rightarrow [\alpha_6], \emptyset}{(\text{AxSK})}, \frac{\Gamma'_1 \vdash (g \text{ 'c'}) :: \alpha_7, \{\alpha_8 \rightarrow [\alpha_8] \dot{=} \text{Char} \rightarrow \alpha_7\}}{(\text{RAPP})}}{\Gamma'_1 \vdash (g (g \text{ 'c'})) :: \alpha_4, \{\alpha_8 \rightarrow [\alpha_8] \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6] \dot{=} \alpha_7 \rightarrow \alpha_4\}}{(\text{RAPP})}}{\Gamma'_1 \vdash \text{Cons } x (g (g \text{ 'c'})) :: \alpha_2, \{\alpha_8 \rightarrow [\alpha_8] \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6] \dot{=} \alpha_7 \rightarrow \alpha_4 \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}}{(\text{SKREK})}}{\Gamma_1 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = [\text{Char}] \rightarrow [[\text{Char}]]}
 \end{array}$$

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 $\{\alpha_8 \rightarrow [\alpha_8] \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6] \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}$ ist.

Daher ist $\Gamma_2 = \Gamma \cup \{g :: [\text{Char}] \rightarrow [[\text{Char}]]\}$.

Satz

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.

$$\Gamma = \{\mathbf{Cons} :: \forall a.a \rightarrow [a] \rightarrow [a], \mathbf{Nil} : \forall a.a\}.$$

$$\Gamma_0 = \Gamma \cup \{f :: \forall \alpha.\alpha, g :: \forall \alpha.\alpha\}$$

$$\frac{\frac{\text{(RAPP)} \frac{\text{(AXK)} \frac{}{\Gamma_0 \vdash \mathbf{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \text{(AXSK)} \frac{}{\Gamma_0 \vdash g :: \alpha_5}}{\Gamma_0 \vdash (\mathbf{Cons} \ g) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} \alpha_5 \rightarrow \alpha_3\}}, \text{(AXK)} \frac{}{\Gamma_0 \vdash \mathbf{Nil} :: [\alpha_2], \emptyset}}{\Gamma_0 \vdash [g] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} \alpha_5 \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}}}{\Gamma_0 \vdash_T f :: \sigma(\alpha_1) = [\alpha_5]}$$

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

Nichtterminierung des iterativen Verfahrens (2)

$$\begin{array}{c}
 \text{(AxK)} \frac{}{\Gamma_0 \vdash \mathbf{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad \text{(AxSK)} \frac{}{\Gamma_0 \vdash \mathbf{f} :: \alpha_5} \\
 \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\mathbf{Cons} \mathbf{f}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} \alpha_5 \rightarrow \alpha_3\}}, \quad \text{(AxK)} \frac{}{\Gamma_0 \vdash \mathbf{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{\Gamma_0 \vdash [\mathbf{f}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} \alpha_5 \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SKREK)} \frac{}{\Gamma_0 \vdash_T \mathbf{g} :: \sigma(\alpha_1) = [\alpha_5]} \\
 \sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \rightarrow [\alpha_5], \alpha_4 \mapsto \alpha_5\} \text{ ist} \\
 \text{Lösung von } \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} \alpha_5 \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}
 \end{array}$$

Daher ist $\Gamma_1 = \Gamma \cup \{\mathbf{f} :: \forall a.[a], \mathbf{g} :: \forall a.[a]\}$. Da $\Gamma_1 \neq \Gamma_0$ muss man weiter iterieren.

Nichtterminierung des iterativen Verfahrens (3)

$$\begin{array}{c}
 \text{(AXK)} \frac{}{\Gamma_1 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad \text{(AXSK)} \frac{}{\Gamma_1 \vdash \mathbf{g} :: [\alpha_5]} \\
 \text{(RAPP)} \frac{}{\Gamma_1 \vdash (\text{Cons } \mathbf{g}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3\}}, \quad \text{(AXK)} \frac{}{\Gamma_1 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{\Gamma_1 \vdash [\mathbf{g}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SKREK)} \frac{}{\Gamma_1 \vdash_T \mathbf{f} :: \sigma(\alpha_1) = [[\alpha_5]]}
 \end{array}$$

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

$$\begin{array}{c}
 \text{(AXK)} \frac{}{\Gamma_1 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \quad \text{(AXSK)} \frac{}{\Gamma_1 \vdash \mathbf{f} :: [\alpha_5]} \\
 \text{(RAPP)} \frac{}{\Gamma_1 \vdash (\text{Cons } \mathbf{f}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3\}}, \quad \text{(AXK)} \frac{}{\Gamma_1 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{\Gamma_1 \vdash [\mathbf{f}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \dot{=} [\alpha_5] \rightarrow \alpha_3, \alpha_3 \dot{=} [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SKREK)} \frac{}{\Gamma_1 \vdash_T \mathbf{g} :: \sigma(\alpha_1) = [[\alpha_5]]}
 \end{array}$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \rightarrow [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ 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 $\Gamma_2 = \Gamma \cup \{\mathbf{f} :: \forall a. [[a]], \mathbf{g} :: \forall a. [[a]]\}$. Da $\Gamma_2 \neq \Gamma_1$ muss man weiter iterieren.

Nichtterminierung des iterativen Verfahrens (4)

Vermutung: Terminiert nicht

Beweis: (Induktion) betrachte den i . Schritt:

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

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

$\Gamma_i \vdash_T f :: \sigma(\alpha_1) = [[\alpha_5]^i]$
 $\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \rightarrow [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ ist
 Lösung von $\{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}$

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

$\Gamma_i \vdash_T g :: \sigma(\alpha_1) = [[\alpha_5]^i]$
 $\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \rightarrow [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ 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. $\Gamma_{i+1} = \Gamma \cup \{f :: \forall a.[a]^{i+1}, g :: \forall a.[a]^{i+1}\}$.

Daher ...

Satz

Das iterative Typisierungsverfahren terminiert nicht immer.

Daher ...

Satz

Das iterative Typisierungsverfahren terminiert nicht immer.

Es gilt sogar:

Theorem

Die iterative Typisierung ist unentscheidbar.

Dies folgt aus der Unentscheidbarkeit der so genannten Semi-Unifikation von First-Order Termen.

- Das iterative Verfahren benötigt die Information aus der Aufrufhierarchie nicht:
- Es liefert die gleichen Typen, unabhängig davon, in welcher Reihenfolge man die Superkombinatoren typisiert.

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

- Die Typisierung bleibt unter Reduktion erhalten („**Type Preservation**“)
Genauer: Für einen Grundtyp τ : $t :: \tau$ vor der Reduktion $t \rightarrow t'$, dann auch $t' :: \tau$ danach.
D.h. polymorphe Typen können auch allgemeiner werden.
- Getypte geschlossene Ausdrücke sind reduzibel, solange sie keine WHNF sind („**Progress Lemma**“).

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, g.d.w.:

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

Lemma (Type Preservation)

Sei s ein wohl-getypter, geschlossener KFPTS+seq-Ausdruck und $s \xrightarrow{\text{name}} s'$. 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 call-by-name reduzibel, d.h. $s \xrightarrow{\text{name}} s'$.

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.

Theorem

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

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

Milner-Schritt: Typisiere SK_1, \dots, SK_m auf einmal, mit der Annahme:
 $\Gamma_M = \Gamma \cup \{SK_1 :: \tau_1, \dots, SK_m :: \tau_m\}$; **ohne Quantoren**
und der Regel: (nächste Folie)

Erzwingen der Terminierung (2)

für $i = 1, \dots, m$:

$$(SKREKM) \frac{\Gamma_M \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau'_i, E_i}{\Gamma_M \vdash_T \text{ für } i = 1, \dots, m \ SK_i :: \sigma(\alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau'_i)}$$

wenn σ Lösung von $E_1 \cup \dots \cup E_m \cup \bigcup_{i=1}^m \{\tau_i \doteq \alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau'_i\}$

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

...

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

die Definitionen von SK_1, \dots, SK_m sind

Als zusätzliche Regel muss im Typisierungsverfahren hinzugefügt werden:

$$(AxSK2) \frac{}{\Gamma \cup \{SK :: \tau\} \vdash SK :: \tau}$$

wenn τ nicht allquantifiziert ist

Erzwingen der Terminierung (3)

Unterschied zum iterativen Schritt:

- Die Typen der zu typisierenden SKs werden nicht allquantifiziert.
- Daher sind während der Typisierung **keine Kopien** dieser 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.
- Daher sind während der Typisierung **keine Kopien** dieser 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 Milner-Schritt terminiert das Verfahren sofort.

Das Milner-Typisierungsverfahren

Milner-Typisierung ist analog zum 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.

Das Milner-Typisierungsverfahren

Milner-Typisierung ist analog zum 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.

Haskell verwendet das Milner-Typisierungs-Verfahren.

Das Milner-Typisierungsverfahren, genauer

Milner-Typisierungsverfahren: SK_1, \dots, SK_m alle SKs einer Äquivalenzklasse bzgl. \simeq
wobei alle kleineren SKs bereits getypt

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

für $i = 1, \dots, m$:

$$\Gamma \cup \{SK_1 :: \beta_1, \dots, SK_m :: \beta_m\}$$

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

wenn σ 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 Milner-typisierbar

Das Milner-Typisierungsverfahren (2)

Vereinfachung: Regel für einen rekursiven SK

$$\text{(MSKREK1)} \frac{\Gamma \cup \{SK :: \beta, x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SK :: \sigma(\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau)}$$

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

Eigenschaften des Milner-Typcheck

Für das Milner-Typisierungsverfahren gelten die folgenden Eigenschaften:

- Das Verfahren **terminiert**.
- Das Verfahren liefert eindeutige Typen (bis auf Umbenennung von Variablen)
- Die Milner-Typisierung ist **entscheidbar**.
- Das Problem, ob ein Ausdruck 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 Milner-typisierbar sein.
- Das 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

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 2^6 .

Verallgemeinert man das Beispiel, dann sind es 2^n .

Beispiele: map

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

$\Gamma_0 = \{\text{Cons} :: \forall a.a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a.[a]\}$

Sei $\Gamma = \Gamma_0 \cup \{\text{map} :: \beta, f :: \alpha_1, xs :: \alpha_2\}$ und $\Gamma' = \Gamma \cup \{y : \alpha_3, ys :: \alpha_4\}$.

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

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) $\frac{(\text{AxV})}{\Gamma \vdash xs :: \alpha_2, \emptyset}$

D.h. $\tau_1 = \alpha_2$ und $E_1 = \emptyset$.

(b) $\frac{(\text{AxK})}{\Gamma \vdash \text{Nil} :: [\alpha_5], \emptyset}$

D.h. $\tau_2 = [\alpha_5]$ und $E_2 = \emptyset$

(c) $\frac{\frac{(\text{AxK})}{\Gamma' \vdash \text{Cons} :: \alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6]}, \frac{(\text{AxV})}{\Gamma' \vdash y :: \alpha_3, \emptyset}}{\frac{(\text{RAPP})}{\Gamma' \vdash (\text{Cons } y) :: \alpha_7, \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \dot{=} \alpha_3 \rightarrow \alpha_7\}}, \frac{(\text{AxV})}{\Gamma' \vdash ys :: \alpha_4, \emptyset}}{\frac{(\text{RAPP})}{\Gamma' \vdash (\text{Cons } y \text{ } ys) :: \alpha_8, \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \dot{=} \alpha_3 \rightarrow \alpha_7, \alpha_7 \dot{=} \alpha_4 \rightarrow \alpha_8\}}}$

D.h. $\tau_3 = \alpha_8$ und $E_3 = \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \dot{=} \alpha_3 \rightarrow \alpha_7, \alpha_7 \dot{=} \alpha_4 \rightarrow \alpha_8\}$

(d) $\frac{(\text{AxK})}{\Gamma \vdash \text{Nil} :: [\alpha_9], \emptyset}$

D.h. $\tau_4 = [\alpha_9]$ und $E_4 = \emptyset$.

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

Beispiele: erneute Betrachtung

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

Iteratives Verfahren liefert Fail nach mehreren Iteration.

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

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

$$\begin{array}{c}
 \text{(AxK)} \frac{}{\Gamma \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, \text{(AxV)} \frac{}{\Gamma \vdash x :: \alpha, \emptyset}, \text{(AxSK2)} \frac{}{\Gamma \vdash g :: \beta, \emptyset}, \text{(AxK)} \frac{}{\Gamma \vdash 'c' :: \text{Char}, \emptyset}, \\
 \text{(RAPP)} \frac{}{\Gamma \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha \rightarrow \alpha_3}, \text{(RAPP)} \frac{}{\Gamma \vdash (g\ (g\ 'c')) :: \alpha_4, \{\beta \dot{=} \text{Char} \rightarrow \alpha_7, \beta \dot{=} \alpha_7 \rightarrow \alpha_4\}} \\
 \text{(RAPP)} \frac{}{\Gamma \vdash \text{Cons } x\ (g\ (g\ 'c')) :: \alpha_2, \{\beta \dot{=} \text{Char} \rightarrow \alpha_7, \beta \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}} \\
 \text{(MSKREK)} \frac{}{\Gamma \vdash_T g :: \sigma(\alpha \rightarrow \alpha_2)}
 \end{array}$$

wobei σ die Lösung von

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

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

Beispiele: erneute Betrachtung (2)

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

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

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

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

wobei σ die Lösung von

$\{\beta \dot{=} \mathbf{Char} \rightarrow \alpha_7, \beta \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \mathbf{Int} \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2, \beta \dot{=} \alpha \rightarrow \alpha_2\}$ ist.

Die Unifikation schlägt fehl, da $[\alpha_5] \dot{=} \mathbf{Char}$ 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)
```

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

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

Grund:

Iteratives Verfahren erlaubt Kopien des Typs für `g`, Milner nicht.

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

Milner Typisierung und Type Safety (2)

- Type-Preservation: Gilt in KFPTSP+seq, aber vermutlich nicht in Haskell:

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

ist 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 Milner-typisierbar

Milner Typisierung und Type Safety (2)

Milner-typisierbar:

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

nicht Milner typisierbar (aber iterativ typisierbar):

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

- Der Effekt kommt von der Allquantifizierung nach erfolgreicher Typisierung:

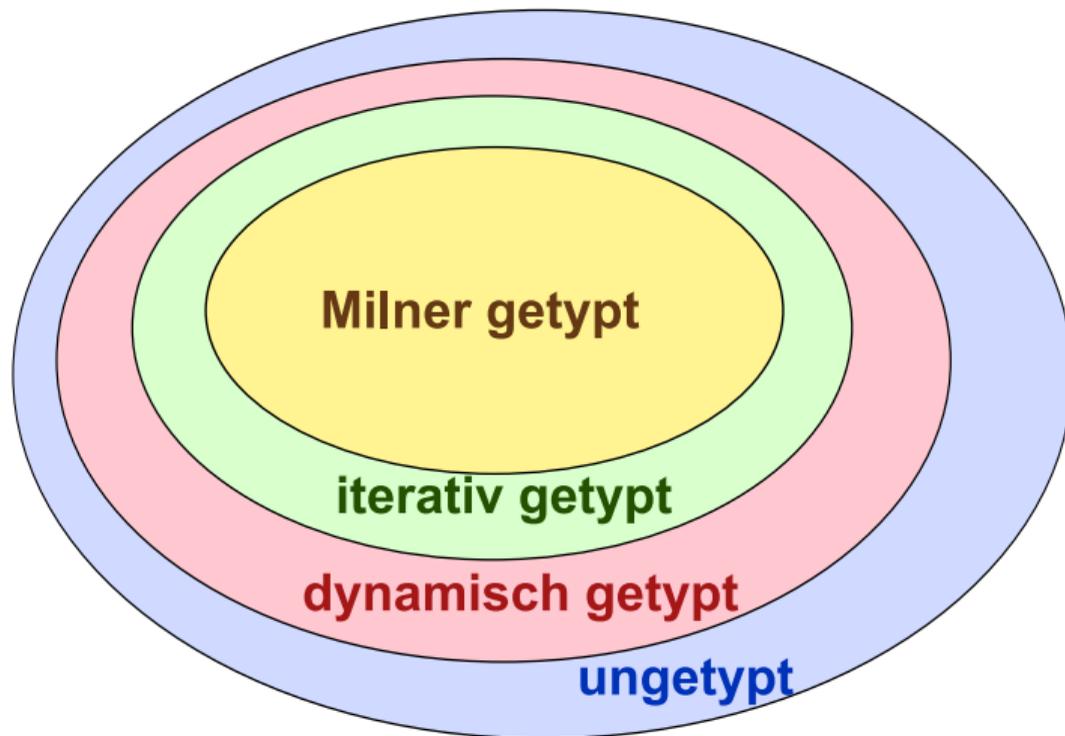
Vorher: einmal kann allquantifiziert werden

Nachher: alles wird auf einmal typisiert.

Das Beispiel ist aber unkritisch, denn:

- Type-Preservation gilt für das iterative Verfahren;
- typisierte Programm sind dynamisch getypt;
- 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 `\x -> const (x True) (x '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 imprädikativen 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 `\x -> const (x True) (x 'A')` zu typisieren:

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

Mit imprädikativen 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!

Beispiel mit Erweiterung: RankNTypes

```
*> :set -XRankNTypes
```

```
*> :t ((\x -> const (x True) (x 'A'))::(forall b.(forall a. a -> b) -> b))
```

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

```
*> :t ((\x -> const (x True) (x 'A'))::(forall b.(forall a. a -> b) -> b)) (const "Hallo")
```

```
((\x -> const (x True) (x 'A'))::(forall b.(forall a. a -> b) -> b)) (const "Hallo") :: [Char]
```

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

Normale Typen sind Rank-1-Typen:

- In $a \rightarrow b \rightarrow a$ sind alle Typvariablen allquantifiziert. D.h. der Typ ist äquivalent zu $\text{forall } a\ b. a \rightarrow b \rightarrow a$
- `forall` in rechten Seiten des Funktionspfeils darf man nach oben schieben.
- D.h. z.B. $\text{forall } a. a \rightarrow (\text{forall } b. b \rightarrow a)$ ist äquivalent zu $\text{forall } a\ b. a \rightarrow b \rightarrow a$ und damit auch Rank-1

RankNTypes

Normale Typen sind Rank-1-Typen:

- In $a \rightarrow b \rightarrow a$ sind alle Typvariablen allquantifiziert. D.h. der Typ ist äquivalent zu $\text{forall } a \ b. \ a \rightarrow b \rightarrow a$
- `forall` in rechten Seiten des Funktionspfeils darf man nach oben schieben.
- D.h. z.B. $\text{forall } a. \ a \rightarrow (\text{forall } b. \ b \rightarrow a)$ ist äquivalent zu $\text{forall } a \ b. \ a \rightarrow b \rightarrow a$ und damit auch Rank-1

Rank-N-Typen

- haben `forall` auch **links** vom Funktionspfeil
- dies darf man **nicht** hochschieben
- N ist die Anzahl der Schachtelungen
- Z.B. $\text{forall } b. (\text{forall } a. \ a \rightarrow b) \rightarrow b$ ist Rank-2
- $(\text{forall } a. \ a \rightarrow a) \rightarrow (\text{forall } b. \ b \rightarrow b)$ ist auch Rank-2
- $((\text{forall } a. \ a \rightarrow a) \rightarrow \text{Int}) \rightarrow \text{Bool} \rightarrow \text{Bool}$ ist Rank-3,
(nicht 4, da gleich zu $((\text{forall } a. \ a \rightarrow a) \rightarrow \text{Int}) \rightarrow (\text{Bool} \rightarrow \text{Bool}))$)