

Einführung in die Funktionale Programmierung:

Typisierung

Prof Dr. Manfred Schmidt-Schauß

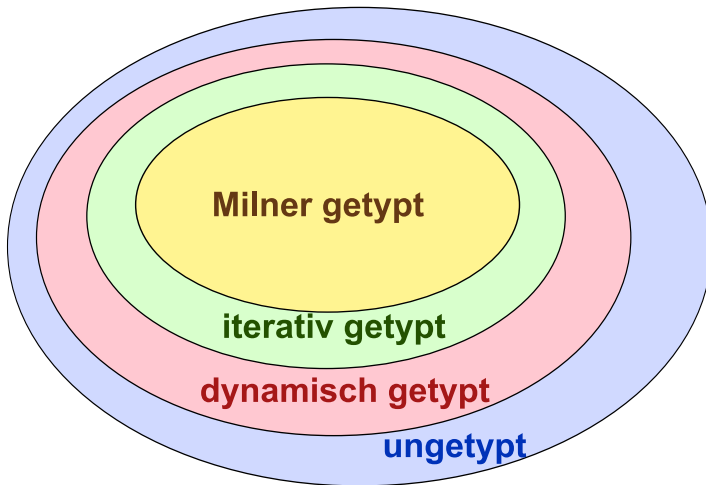
WS 2022/23

- 1 Motivation
- 2 Typen: Sprechweisen, Notationen und Unifikation
- 3 Typisierungsverfahren
 - Iteratives Typisierungsverfahren
 - Das Hindley-Milner-Typisierungsverfahren
- 4 Typklassen

Ziele des Kapitels

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

KFPTS+seq



Motivation

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

Motivation (2)

Minimalanforderungen:

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

Motivation (2)

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.
- Typsysteme mit **dependent types** sind aktuell im Fokus der Forschung; und werden in Haskell-ähnlichen Programmiersprachen erprobt.
Diese sind komplexer als polymorphe Typisierung.

Naiver Ansatz: KFPTS+seq

Naive Definition von „korrekt getypt“:

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

Naiver Ansatz: KFPTS+seq

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)

(Haskell-Programm auf der Webseite, Archiv?)

Unentscheidbarkeit der dynamischen Typisierung (2)

Für eine TM-Beschreibung b und Eingabe e sei

```
 $s :=$  if tmEncode  $b$   $e$   
    then caseBool Nil of {True  $\rightarrow$  True; False  $\rightarrow$  False}  
    else caseBool Nil of {True  $\rightarrow$  True; False  $\rightarrow$  False}
```

Es gilt:

s ist genau dann dynamisch ungetypt, wenn die 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.

Typen

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 Grundtypen, aber keine Basistypen.
- **[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 die Reihenfolge der α_i egal ist, 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

Eine **Typsubstitution** ist eine 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

$$\sigma_E(TV) := \sigma(TV), \text{ falls } \sigma \text{ die Variable } TV \text{ abbildet}$$

$$\sigma_E(TV) := TV, \text{ falls } \sigma \text{ die Variable } TV \text{ nicht abbildet}$$

$$\sigma_E(TC \ T_1 \ \dots \ T_n) := TC \ \sigma_E(T_1) \ \dots \ \sigma_E(T_n)$$

$$\sigma_E(T_1 \rightarrow T_2) := \sigma_E(T_1) \rightarrow \sigma_E(T_2)$$

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

Semantik eines polymorphen Typs

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

Regel für Anwendung ($s\ t$):

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

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**

Typregeln

Regel für Anwendung $(s\ t)$:

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

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**

Unifikationsproblem

Definition

Ein **Unifikationsproblem** auf Typen ist gegeben durch eine Menge Γ von Gleichungen der Form $\tau_1 \doteq \tau_2$, wobei τ_1 und τ_2 polymorphe Typen sind.

Eine **Lösung** eines Unifikationsproblem Γ 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 Γ ist ein Unifikator σ , so dass gilt: Für jeden anderen Unifikator ρ von Γ gibt es eine Substitution γ so dass $\rho(x) = \gamma \circ \sigma(x)$ für alle $x \in FV(\Gamma)$.

Unifikationsalgorithmus

- Datenstruktur: Γ = Multimenge von Gleichungen

Multimenge $\equiv$ "Menge" mit mehrfachem Vorkommen von Elementen

- $\Gamma \cup \Gamma'$ sei die **disjunkte** Vereinigung von zwei Multimengen
- $\Gamma[\tau/\alpha]$ ist definiert als $\{s[\tau/\alpha] \doteq t[\tau/\alpha] \mid (s \doteq t) \in \Gamma\}$.

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

- Fail auftritt, oder
- keine Regel mehr anwendbar ist

Unifikationsalgorithmus: Schlussregeln (1)

Dekomposition:

$$\text{DECOMPOSE1} \frac{\Gamma \cup \{TC \ \tau_1 \ \dots \ \tau_n \dot{=} TC \ \tau'_1 \ \dots \ \tau'_n\}}{\Gamma \cup \{\tau_1 \dot{=} \tau'_1, \dots, \tau_n \dot{=} \tau'_n\}}$$

$$\text{DECOMPOSE2} \frac{\Gamma \cup \{\tau_1 \rightarrow \tau_2 \dot{=} \tau'_1 \rightarrow \tau'_2\}}{\Gamma \cup \{\tau_1 \dot{=} \tau'_1, \tau_2 \dot{=} \tau'_2\}}$$

Unifikationsalgorithmus: Schlussregeln (2)

Orientierung, Elimination:

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

wenn τ_1 keine Typvariable und α Typvariable

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

wobei α Typvariable

Unifikationsalgorithmus: Schlussregeln (3)

Einsetzung, Occurs-Check:

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

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

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

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

Unifikationsalgorithmus: Abbruchregeln

Fail-Regeln:

$$\text{FAIL1} \frac{\Gamma \cup \{(TC_1 \ \tau_1 \ \dots \ \tau_n) \dot{=} (TC_2 \ \tau'_1 \ \dots \ \tau'_m)\}}{\text{Fail}} \\ \text{wenn } TC_1 \neq TC_2$$

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

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

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: $\{(a \rightarrow b) \dot{=} \text{Bool} \rightarrow \text{Bool}\}$:

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

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

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

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

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

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

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

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: $\{(a \rightarrow b) \dot{=} \text{Bool} \rightarrow \text{Bool}\}$:

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

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

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

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

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

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** gdw. es **keinen** Unifikator für die Eingabe gibt.
- Der Algorithmus endet erfolgreich gdw. es einen Unifikator für die Eingabe gibt. Das Gleichungssystem Γ 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 kann dann abgelesen werden als $\sigma = \{\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n\}$.

- Liefert der Algorithmus **einen** Unifikator, dann ist es **ein allgemeinsten Unifikator**.
 σ allgemeinst bedeutet: jede andere Lösung ist abgedeckt, d.h ist spezieller als σ ,
genauer: kann durch weitere Einsetzung aus σ erzeugt werden.

Eigenschaften des Unifikationsalgorithmus (2)

- 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 (3)

- Die Typen in der Resultat-Substitution können **exponentiell groß** werden. Aber sind **komprimiert polynomiell** groß.
- Der Unifikationsalgorithmus kann aber so implementiert werden, dass er **Zeit $O(n * \log n)$** benötigt. Man muss Sharing dazu beachten; Dazu 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. man kann im wesentlichen alle PTIME-Probleme als Unifikationsproblem darstellen:
Interpretation ist: Unifikation ist nicht effizient parallelisierbar.

Typisierungsverfahren

Wir betrachten nun die

polymorphe Typisierung

von KFPTSP+seq-Ausdrücken

Wir verschieben zunächst: Typisierung von Superkombinatoren

Typisierungsverfahren

Wir betrachten nun die

polymorphe Typisierung

von $\text{KFPTSP} + \text{seq}$ -Ausdrücken

Wir verschieben zunächst: Typisierung von Superkombinatoren

Nächster Schritt:

Typisierungsalgorithmus und Regeln?

Was sind die Typisierungsregeln?

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: map not

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

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: map not

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

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

Anwendungsregel mit Unifikation

Beispiele rechnen: `map length`

Typisierung mit Bindern

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

- Typisiere den Rumpf s
- Wenn x nicht in s , dann typisiere $s :: \tau$
- $\lambda x.s$ hat dann einen Funktionstyp $\alpha \rightarrow \tau$.
- Wenn x im Rumpf s vorkommt, brauchen wir $x : \tau_1$ bei der Berechnung von τ !
- α muss normalerweise verfeinert werden.

Typisierung mit Bindern

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

- Typisiere den Rumpf s
- Wenn x nicht in s , dann typisiere $s :: \tau$
- $\lambda x.s$ hat dann einen Funktionstyp $\alpha \rightarrow \tau$.
- Wenn x im Rumpf s vorkommt, **brauchen wir** $x : \tau_1$ bei der Berechnung von τ !
- α muss normalerweise verfeinert werden.

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:

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

Bedeutung:

Gegeben eine Menge A von Typ-Annahmen.

der Form $s :: \tau$, wobei s Ausdruck, τ Typ ist.

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

- In A kommen nur Typ-Annahmen für Konstruktoren, Variablen, Superkombinatoren vor.
- In E sammeln wir Gleichungen. Diese werden sofort (oder später) unifiziert.
- $\vdash$ symbolisiert den Begriff Herleitung.

Typisierung von Ausdrücken (2)

Herleitungsregeln schreiben wir in der Form

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

$$\frac{A_1 \vdash s_1 :: \tau_1, E_1 \quad \dots \quad A_k \vdash s_k :: \tau_k, E_K}{A \vdash s :: \tau, E}$$

Andere Lesart:

- Wenn man $A \vdash s :: \tau, E$ berechnen will, muss man erst den Teil auf dem Bruchstrich berechnen.
- Die Information kann in beide Richtungen fließen !

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:

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

Typisierungsregeln für KFPTS+seq Ausdrücke (1)

Axiom für Variablen:

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

Axiom für Konstruktoren:

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

wobei β_i neue Typvariablen sind

- Beachte: Bei jeder Typisierung des Konstruktors c wird ein mit neuen Typ-Variablen umbenannter Typ verwendet!

Typisierungsregeln für KFPTS+seq Ausdrücke (2)

Axiom für Superkombinatoren, deren Typ schon bekannt ist:

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

wobei β_i neue Typvariablen sind

- Beachte: Auch hier wird jedesmal ein mit neuen Variablen umbenannter Typ verwendet!

Typisierungsregeln für KFPTS+seq Ausdrücke (3)

Regel für Anwendungen:

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

wobei α neue Typvariable

Typisierungsregeln für KFPTS+seq Ausdrücke (3)

Regel für Anwendungen:

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

wobei α neue Typvariable

Regel für seq:

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

Typisierungsregeln für KFPTS+seq Ausdrücke (4)

Regel für Abstraktionen:

$$(\text{RAbs}) \frac{A \cup \{x :: \alpha\} \vdash s :: \tau, E}{A \vdash \lambda x. s :: \alpha \rightarrow \tau, E}$$

wobei α eine neue Typvariable

In dieser Regel werden die Annahmen erweitert .

In E stehen die Bedingungen (Gleichungen) zu den Typvariablen.

α wird normalerweise eingeschränkt.

Typisierungsregeln für KFPS+seq Ausdrücke (5)

Typisierung eines case: Prinzipien

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

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

Typisierungsregeln für $KFPTS+seq$ Ausdrücke (6)

RCase ist die Regel für case:

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

für alle $i = 1, \dots, m$:

$A \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,\text{ar}(c_i)} :: \alpha_{i,\text{ar}(c_i)}\} \vdash (c_i \ x_{i,1} \ \dots \ x_{i,\text{ar}(c_i)}) :: \tau_i, E_i$

für alle $i = 1, \dots, m$:

$A \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,\text{ar}(c_i)} :: \alpha_{i,\text{ar}(c_i)}\} \vdash t_i :: \tau'_i, E'_i$

(RCASE)

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

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

Typisierungsalgorithmus für KFPTS+seq-Ausdrücke

Algorithmus:

Sei s ein geschlossener KFPTS+seq-Ausdruck, wobei die Typen für alle in s benutzten Superkombinatoren und Konstruktoren bekannt sind. (d.h. diese Typen sind schon berechnet oder vorgegeben)

- 1 Starte mit Anfangsannahme A , die Typen für die Konstruktoren und die Superkombinatoren enthält.
- 2 Leite $A \vdash s :: \tau, E$ 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
(Oder Abbrechen mit Fail):

Typberechnung:

$$(\text{RUNIF}) \frac{A \vdash s :: \tau, E}{A \vdash s :: \sigma(\tau), E_\sigma}$$

wobei E_σ das gelöste Gleichungssystem zu E ist
und σ der ablesbare Unifikator ist

Wohlgetyptheit

Definition

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

(Typisierung von Superkombinatoren kommt noch)

(Schwieriger!, da diese rekursiv sein können)

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\text{(RAPP)} \frac{A_0 \vdash (\text{Cons True}) :: \tau_1, E_1, \quad A_0 \vdash \text{Nil} :: \tau_2, E_2}{A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha_4\}}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\frac{A_0 \vdash (\text{Cons True}) :: \tau_1, E_1, \quad (\text{AxK}) \quad \overline{A_0 \vdash \text{Nil} :: [\alpha_3], \emptyset}}{(\text{RAPP}) \quad A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, E_1 \cup \emptyset \cup \{\tau_1 \doteq [\alpha_3] \rightarrow \alpha_4\}}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\begin{array}{c}
 \text{(RAPP)} \frac{A_0 \vdash \text{Cons} :: \tau_3, E_3, \quad A_0 \vdash \text{True} :: \tau_4, E_4}{A_0 \vdash (\text{Cons True}) :: \alpha_2, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_2\} \cup E_3 \cup E_4}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\
 \text{(RAPP)} \frac{}{A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_2\} \cup E_3 \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}
 \end{array}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset, \quad A_0 \vdash \text{True} :: \tau_4, E_4} \\
 \text{(RAPP)} \frac{}{A_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \tau_4 \rightarrow \alpha_2\} \cup E_4, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\
 \text{(RAPP)} \frac{}{A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \tau_4 \rightarrow \alpha_2\} \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}
 \end{array}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{True} :: \text{Bool}, \emptyset} \\
 \text{(RAPP)} \frac{}{A_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\
 \text{(RAPP)} \frac{}{A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\} \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}
 \end{array}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{True} :: \text{Bool}, \emptyset} \\
 \text{(RApp)} \frac{}{A_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\
 \text{(RApp)} \frac{}{A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}
 \end{array}$$

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{True} :: \text{Bool}, \emptyset} \\
 \text{(RApp)} \frac{}{A_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\
 \text{(RApp)} \frac{}{A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}
 \end{array}$$

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: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{True} :: \text{Bool}, \emptyset} \\
 \text{(RApp)} \frac{}{A_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\
 \text{(RApp)} \frac{}{A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}
 \end{array}$$

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

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: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

α : Variablen; τ : Typen und E Gleichungsmengen, die berechnet werden.

$$\begin{array}{c}
 \text{(AxK)} \frac{}{A_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{True} :: \text{Bool}, \emptyset} \\
 \text{(RAPP)} \frac{}{A_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}}, \quad \text{(AxK)} \frac{}{A_0 \vdash \text{Nil} :: [\alpha_3], \emptyset} \\
 \text{(RAPP)} \frac{}{A_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2, \alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}
 \end{array}$$

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

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 (Cons True Nil)

Vereinfachung für die Intuition und zum selbst Nachrechnen:

Typisierung von Cons True Nil

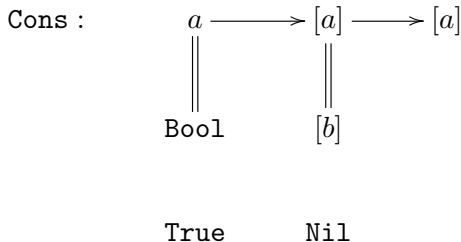
Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

Beispiele: Typisierung von (Cons True Nil)

Vereinfachung für die Intuition und zum selbst Nachrechnen:

Typisierung von Cons True Nil

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$



Beispiele: Typisierung von (Cons True Nil)

Vereinfachung für die Intuition und zum selbst Nachrechnen:

Typisierung von Cons True Nil

Anfangsannahme: $A_0 = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a. [a], \text{True} :: \text{Bool}\}$

Cons :

$$\begin{array}{c}
 a \longrightarrow [a] \longrightarrow [a] \\
 \parallel \qquad \qquad \parallel \\
 \text{Bool} \qquad \qquad [b]
 \end{array}$$

Ergibt: True Nil
 $a = b = \text{Bool}$ und
 Ergebnis ist $[a]$ d.h. (Cons True Nil):[Bool]

Beispiele: Typisierung von $\lambda x.x$

Typisierung von $\lambda x.x$

Starte mit: Anfangsannahme: $A_0 = \emptyset$

$$\text{(RAbs)} \quad \frac{A_0 \cup \{x :: \alpha\} \vdash x :: \tau, E}{A_0 \vdash (\lambda x.x) :: \alpha \rightarrow \tau, E}$$

Beispiele: Typisierung von $\lambda x.x$

Typisierung von $\lambda x.x$

Starte mit: Anfangsannahme: $A_0 = \emptyset$

$$\begin{array}{c} \text{(AxV)} \frac{}{A_0 \cup \{x :: \alpha\} \vdash x :: \alpha, \emptyset} \\ \text{(RAbs)} \frac{}{A_0 \vdash (\lambda x.x) :: \alpha \rightarrow \alpha, \emptyset} \end{array}$$

Beispiele: Typisierung von $\lambda x.x$

Typisierung von $\lambda x.x$

Starte mit: Anfangsannahme: $A_0 = \emptyset$

$$\begin{array}{c} \text{(AxV)} \frac{}{A_0 \cup \{x :: \alpha\} \vdash x :: \alpha, \emptyset} \\ \text{(RAbs)} \frac{}{A_0 \vdash (\lambda x.x) :: \alpha \rightarrow \alpha, \emptyset} \end{array}$$

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: $A_0 = \emptyset$

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

Beispiele: Typisierung von Ω

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

Starte mit: Anfangsannahme: $A_0 = \emptyset$

$$\begin{array}{c}
 \text{(R}_{\text{ABS}}\text{)} \frac{\{x :: \alpha_2\} \vdash (x\ x) :: \tau_6, E_1}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \rightarrow \tau_6, E_1} \ , \ \emptyset \vdash (\lambda y.(y\ y)) :: \tau_2, E_2 \\
 \text{(R}_{\text{APP}}\text{)} \frac{}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, E_1 \cup E_2 \cup \{\alpha_2 \rightarrow \tau_6 \doteq \tau_2 \rightarrow \alpha_1\}}
 \end{array}$$

Beispiele: Typisierung von Ω

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

Starte mit: Anfangsannahme: $A_0 = \emptyset$

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

Beispiele: Typisierung von Ω

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

Starte mit: Anfangsannahme: $A_0 = \emptyset$

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

Beispiele: Typisierung von Ω

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

Starte mit: Anfangsannahme: $A_0 = \emptyset$

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

Beispiele: Typisierung von Ω

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

Starte mit: Anfangsannahme: $A_0 = \emptyset$

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

Beispiele: Typisierung von Ω

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

Starte mit: Anfangsannahme: $A_0 = \emptyset$

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

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

Vereinfachung für die Intuition und zum selbst Nachrechnen

Typisierung von $\lambda x.x\ x$

Typisierung kann in $\lambda x.(x\ x)$ das x nur **monomorph** typisieren

Betrachte die Anwendung $(x\ x)$:

Beispiele: Typisierung von $\lambda x.(x\ x)$

Vereinfachung für die Intuition und zum selbst Nachrechnen

Typisierung von $\lambda x.x\ x$

Typisierung kann in $\lambda x.(x\ x)$ das x nur **monomorph** typisieren

Betrachte die Anwendung $(x\ x)$:

$$\begin{array}{ccc}
 x : & a & \longrightarrow b \\
 & \parallel & \\
 & (a \rightarrow b) &
 \end{array}$$

x

Beispiele: Typisierung von $\lambda x.(x\ x)$

Vereinfachung für die Intuition und zum selbst Nachrechnen

Typisierung von $\lambda x.x\ x$

Typisierung kann in $\lambda x.(x\ x)$ das x nur **monomorph** typisieren

Betrachte die Anwendung $(x\ x)$:

$$\begin{array}{ccc}
 x : & a & \longrightarrow b \\
 & \parallel & \\
 & (a \rightarrow b) &
 \end{array}$$

x

Erfordert: Lösung von $a = (a \rightarrow b)$ aber: hat keine Lösung !,

da das x in $\lambda x.(x\ x)$ monomorph typisiert werden muss.

Beispiele: Typisierung von $\lambda x.(x\ x)$

Vereinfachung für die Intuition und zum selbst Nachrechnen

Typisierung von $\lambda x.x\ x$

Typisierung kann in $\lambda x.(x\ x)$ das x nur **monomorph** typisieren

Betrachte die Anwendung $(x\ x)$:

$$\begin{array}{ccc}
 x : & a & \longrightarrow b \\
 & \parallel & \\
 & (a \rightarrow b) &
 \end{array}$$

x

Erfordert: Lösung von $a = (a \rightarrow b)$ aber: hat keine Lösung !,

da das x in $\lambda x.(x\ x)$ monomorph typisiert werden muss.

Vorsicht: (map map) hat Typ $[a \rightarrow b] \rightarrow [[a] \rightarrow [b]]$

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:

$$\begin{aligned} A_0 = \{ & \text{map} :: \forall a, b. (a \rightarrow b) \rightarrow [a] \rightarrow [b], \\ & \text{length} :: \forall a. [a] \rightarrow \text{Int}, \\ & \text{Nil} :: \forall a. [a] \\ & \text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a] \\ & \} \end{aligned}$$

$$\begin{array}{c} \text{(AxK)} \overline{B_8}, \text{(AxV)} \overline{B_9} \\ \text{(RAPP)} \frac{\overline{B_8}, \overline{B_9}}{B_6}, \text{(AxV)} \overline{B_7} \\ \text{(RCASE)} \frac{\text{(AxV)} \overline{B_3}, \text{(AxK)} \overline{B_4}, \text{(RAPP)} \frac{\overline{B_8}, \overline{B_9}}{B_6}, \text{(AxV)} \overline{B_7}}{B_5}, \text{(AxK)} \overline{B_{10}}, \text{(RAPP)} \frac{\text{(AxSK)} \overline{B_{14}}, \text{(AxSK)} \overline{B_{15}}}{B_{12}}, \text{(AxV)} \overline{B_{13}} \\ \text{(RABS)} \frac{\text{(RCASE)} \frac{\text{(AxV)} \overline{B_3}, \text{(AxK)} \overline{B_4}, \text{(RAPP)} \frac{\overline{B_8}, \overline{B_9}}{B_6}, \text{(AxV)} \overline{B_7}}{B_5}, \text{(AxK)} \overline{B_{10}}, \text{(RAPP)} \frac{\text{(AxSK)} \overline{B_{14}}, \text{(AxSK)} \overline{B_{15}}}{B_{12}}, \text{(AxV)} \overline{B_{13}}}{B_2} \\ B_1 \end{array}$$
$$\begin{aligned}
B_1 = \quad & A_0 \vdash t :: \alpha_1 \rightarrow \alpha_{13}, \\
& \{ \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_3 \rightarrow \alpha_6, \alpha_6 \dot{=} \alpha_4 \rightarrow \alpha_7, \\
& (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \dot{=} ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \dot{=} \alpha_4 \rightarrow \alpha_{12}, \\
& \alpha_1 \dot{=} [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \dot{=} [\alpha_{14}], \alpha_{13} = \alpha_{12}, \} \\
B_2 = \quad & A_0 \cup \{ xs :: \alpha_1 \} \vdash \\
& \text{case}_{\text{List}} \, xs \text{ of } \{ \mathbf{Nil} \rightarrow \mathbf{Nil}; (\mathbf{Cons} \, y \, ys) \rightarrow \text{map length } ys \} :: \alpha_{13}, \\
& \{ \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_3 \rightarrow \alpha_6, \alpha_6 \dot{=} \alpha_4 \rightarrow \alpha_7, \\
& (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \dot{=} ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \dot{=} \alpha_4 \rightarrow \alpha_{12}, \\
& \alpha_1 \dot{=} [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \dot{=} [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}
\end{aligned}$$

Herleitungsbaum:

$$\begin{array}{c} \text{(AxK)} \overline{B_8}, \text{(AxV)} \overline{B_9} \\ \text{(RAPP)} \frac{\overline{B_8}, \overline{B_9}}{B_6}, \text{(AxV)} \overline{B_7} \\ \text{(RCASE)} \overline{B_3}, \text{(AxK)} \overline{B_4}, \text{(RAPP)} \frac{\overline{B_8}, \overline{B_9}}{B_6}, \text{(AxK)} \overline{B_{10}}, \text{(RAPP)} \frac{\text{(AxSK)} \overline{B_{14}}, \text{(AxSK)} \overline{B_{15}}}{B_{12}}, \text{(AxV)} \overline{B_{13}} \\ \text{(RABS)} \frac{\overline{B_3}, \text{(AxK)} \overline{B_4}, \text{(RAPP)} \frac{\overline{B_8}, \overline{B_9}}{B_6}, \text{(AxK)} \overline{B_{10}}, \text{(RAPP)} \frac{\text{(AxSK)} \overline{B_{14}}, \text{(AxSK)} \overline{B_{15}}}{B_{12}}, \text{(AxV)} \overline{B_{13}}}{B_2} \\ B_1 \end{array}$$

Beschriftungen:

$$\begin{aligned}
B_3 &= A_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset \\
B_4 &= A_0 \cup \{xs :: \alpha_1\} \vdash \mathbf{Nil} :: [\alpha_2], \emptyset \\
B_5 &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\mathbf{Cons} \ y \ ys) :: \alpha_7, \\
&\quad \{\alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7\} \\
B_6 &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\mathbf{Cons} \ y) :: \alpha_6, \\
&\quad \{\alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6\} \\
B_7 &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset \\
B_8 &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \mathbf{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset \\
B_9 &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash y :: \alpha_3, \emptyset \\
B_{10} &= A_0 \cup \{xs :: \alpha_1\} \vdash \mathbf{Nil} :: [\alpha_{14}], \emptyset
\end{aligned}$$

$$\frac{\begin{array}{c} \text{(AxV)} \overline{B_3}, \text{(AxK)} \overline{B_4}, \\ \text{(RCASE)} \end{array}}{\text{(RABS)}} \frac{\begin{array}{c} \text{(AxSK)} \overline{B_8}, \text{(AxV)} \overline{B_9} \\ \text{(RAPP)} \overline{B_6}, \text{(AxV)} \overline{B_7} \\ \text{(AxK)} \overline{B_{10}}, \end{array}}{\begin{array}{c} B_2 \\ B_1 \end{array}} \frac{\begin{array}{c} \text{(AxSK)} \overline{B_{14}}, \text{(AxSK)} \overline{B_{15}} \\ \text{(RAPP)} \overline{B_{12}}, \text{(AxV)} \overline{B_{13}} \end{array}}{B_{11}}$$
$$\begin{aligned}
B_{11} &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\text{map length}) \, ys :: \alpha_{12}, \\
&\quad \{(\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \text{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}\} \\
B_{12} &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\text{map length}) :: \alpha_{11}, \\
&\quad \{(\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \text{Int}) \rightarrow \alpha_{11}\} \\
B_{13} &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset \\
B_{14} &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \text{map} :: (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9], \emptyset \\
B_{15} &= A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \text{length} :: [\alpha_{10}] \rightarrow \text{Int}, \emptyset
\end{aligned}$$

Beispiele: Typisierung eines Ausdrucks mit SKs (5)

Beschriftung unten:

$$\begin{aligned}
 B_1 = \quad & A_0 \vdash t :: \alpha_1 \rightarrow \alpha_{13}, \\
 & \{ \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7, \\
 & (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}, \\
 & \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}
 \end{aligned}$$

Beispiele: Typisierung eines Ausdrucks mit SKs (5)

Beschriftung unten:

$$\begin{aligned}
 B_1 = \quad & A_0 \vdash t :: \alpha_1 \rightarrow \alpha_{13}, \\
 & \{ \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7, \\
 & (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}, \\
 & \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}
 \end{aligned}$$

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:

$$\begin{aligned}
 B_1 = \quad & A_0 \vdash t :: \alpha_1 \rightarrow \alpha_{13}, \\
 & \{ \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7, \\
 & (\alpha_8 \rightarrow \alpha_9) \rightarrow [\alpha_8] \rightarrow [\alpha_9] \doteq ([\alpha_{10}] \rightarrow \mathbf{Int}) \rightarrow \alpha_{11}, \alpha_{11} \doteq \alpha_4 \rightarrow \alpha_{12}, \\
 & \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}
 \end{aligned}$$

Löse mit Unifikation:

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

Ergibt:

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

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

zur Erinnerung:

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

Vereinfachung Typisierung von $(\text{map length } xs)$

für die Intuition und selbst Nachrechnen:

Typisierung von $\text{map length } xs$

Starte mit:

Ann: $A_0 = \{\text{map} :: \forall a, b. (a \rightarrow b) \rightarrow [a] \rightarrow [b], \text{length} :: \forall a. [a] \rightarrow \text{Int}\}$

Vereinfachung Typisierung von $(\text{map length } xs)$

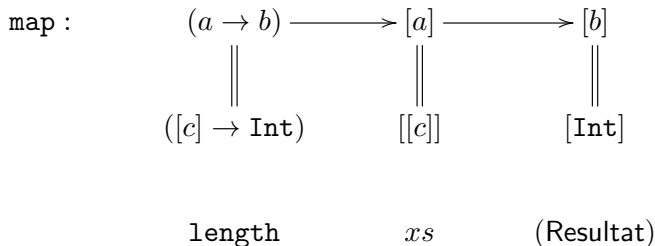
für die Intuition und selbst Nachrechnen:

Typisierung von $\text{map length } xs$

Starte mit:

Ann: $A_0 = \{\text{map} :: \forall a, b. (a \rightarrow b) \rightarrow [a] \rightarrow [b], \text{length} :: \forall a. [a] \rightarrow \text{Int}\}$

Typisiere $\text{map length } xs$



Vereinfachung Typisierung von $(\text{map length } xs)$

für die Intuition und selbst Nachrechnen:

Typisierung von $\text{map length } xs$

Starte mit:

Ann: $A_0 = \{\text{map} :: \forall a, b. (a \rightarrow b) \rightarrow [a] \rightarrow [b], \text{length} :: \forall a. [a] \rightarrow \text{Int}\}$

Typisiere $\text{map length } xs$

map :	$(a \rightarrow b)$	$\longrightarrow$	$[a]$	$\longrightarrow$	$[b]$
	$\parallel$		$\parallel$		$\parallel$
	$([c] \rightarrow \text{Int})$		$[[c]]$		$[\text{Int}]$

length xs (Resultat)

Ergibt: $a = [c]$, $b = \text{Int}$
 $(\text{map length } xs) :: [\text{Int}]$

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

Die Funktion `const` ist definiert als

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

```
const x y = x
```

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

Zum Beispiel: nach Einsetzen von $x = \text{Id}$ wäre der Rumpf getypt.

Anfangsannahme:

$$A_0 = \{\text{const} :: \forall a, b. a \rightarrow b \rightarrow a, \text{True} :: \text{Bool}, \text{'A'} :: \text{Char}\}.$$

Bsp.: Typisierung von Lambda-geb. Variablen (2)

$$\begin{array}{c}
 \frac{}{(AxK) A_1 \vdash \text{const} :: \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_2, \emptyset,} \quad \frac{}{(AxV) A_1 \vdash x :: \alpha_1,} \quad \frac{}{(AxK) A_1 \vdash \text{True} :: \text{Bool}} \\
 \frac{}{(RApp) A_1 \vdash \text{const} (x \text{ True}) :: \alpha_5, E_2} \quad \frac{}{(RApp) A_1 \vdash (x \text{ True}) :: \alpha_4, E_1} \quad \frac{}{(AxV) A_1 \vdash x :: \alpha_1,} \quad \frac{}{(AxK) A_1 \vdash 'A' :: \text{Char}} \\
 \frac{}{(RApp) A_1 \vdash \text{const} (x \text{ True}) (x 'A') :: \alpha_7, E_4} \quad \frac{}{(RApp) A_1 \vdash (x 'A') :: \alpha_6, E_3} \\
 \frac{}{(RAbs) A_0 \vdash \lambda x. \text{const} (x \text{ True}) (x 'A') :: \alpha_1 \rightarrow \alpha_7, E_4}
 \end{array}$$

wobei $A_1 = A_0 \cup \{x :: \alpha_1\}$ und:

$$\begin{aligned}
 E_1 &= \{\alpha_1 \dot{=} \text{Bool} \rightarrow \alpha_4\} \\
 E_2 &= \{\alpha_1 \dot{=} \text{Bool} \rightarrow \alpha_4, \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_2 \dot{=} \alpha_4 \rightarrow \alpha_5\} \\
 E_3 &= \{\alpha_1 \dot{=} \text{Char} \rightarrow \alpha_6\} \\
 E_4 &= \{\alpha_1 \dot{=} \text{Bool} \rightarrow \alpha_4, \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_2 \dot{=} \alpha_4 \rightarrow \alpha_5, \alpha_1 \dot{=} \text{Char} \rightarrow \alpha_6, \\
 &\quad \alpha_5 \dot{=} \alpha_6 \rightarrow \alpha_7\}
 \end{aligned}$$

Bsp.: Typisierung von Lambda-geb. Variablen (2)

$$\begin{array}{c}
 \frac{\frac{(AxK) \overline{A_1 \vdash \text{const} :: \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_2, \emptyset}, \quad \frac{(AxV) \overline{A_1 \vdash x :: \alpha_1}, \quad (AxK) \overline{A_1 \vdash \text{True} :: \text{Bool}}}{(RAPP) \overline{A_1 \vdash (x \text{ True}) :: \alpha_4, E_1}}}{(RAPP) \overline{A_1 \vdash \text{const} (x \text{ True}) :: \alpha_5, E_2}}, \quad \frac{(AxV) \overline{A_1 \vdash x :: \alpha_1}, \quad (AxK) \overline{A_1 \vdash 'A' :: \text{Char}}}{(RAPP) \overline{A_1 \vdash (x 'A') :: \alpha_6, E_3}} \\
 \hline
 (RAPP) \overline{A_1 \vdash \text{const} (x \text{ True}) (x 'A') :: \alpha_7, E_4} \\
 \hline
 (RAbs) \overline{A_0 \vdash \lambda x. \text{const} (x \text{ True}) (x 'A') :: \alpha_1 \rightarrow \alpha_7, E_4}
 \end{array}$$

wobei $A_1 = A_0 \cup \{x :: \alpha_1\}$ und:

$$\begin{aligned}
 E_1 &= \{\alpha_1 \dot{=} \text{Bool} \rightarrow \alpha_4\} \\
 E_2 &= \{\alpha_1 \dot{=} \text{Bool} \rightarrow \alpha_4, \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_2 \dot{=} \alpha_4 \rightarrow \alpha_5\} \\
 E_3 &= \{\alpha_1 \dot{=} \text{Char} \rightarrow \alpha_6\} \\
 E_4 &= \{\alpha_1 \dot{=} \text{Bool} \rightarrow \alpha_4, \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_2 \dot{=} \alpha_4 \rightarrow \alpha_5, \alpha_1 \dot{=} \text{Char} \rightarrow \alpha_6, \\
 &\quad \alpha_5 \dot{=} \alpha_6 \rightarrow \alpha_7\}
 \end{aligned}$$

Die Unifikation schlägt fehl, da $\text{Char} \neq \text{Bool}$

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 (Funktionen) polymorph sind**.
- KFPS+seq hat kein let, aber **Superkombinatoren**, die wie (ein eingeschränkter rekursives) let wirken

Typisierung rekursiver Superkombinatoren

Typisierung rekursiver Superkombinatoren

Rekursive Superkombinatoren

Beispiel

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

Rekursive Superkombinatoren

Definition (direkt rekursiv, rekursiv, verschränkt rekursiv)

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

$$SK_i \preceq SK_j$$

gdw. SK_j den Superkombinator SK_i im Rumpf benutzt.

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

Typisierung von nicht-rekursiven Superkombinatoren

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

Typisierung von nicht-rekursiven Superkombinatoren

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

Typisierungsregel für (geschlossene) nicht-rekursive SK:

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

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

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

Beispiel: Typisierung von (.)

$$(.) \ f \ g \ x = f \ (g \ x)$$

A_0 ist leer, da keine Konstruktoren oder SK vorkommen.

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

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

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

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

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

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

Typisierung von rekursiven Superkombinatoren

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

Typisierung von rekursiven Superkombinatoren

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

Idee des Iterativen Typisierungsverfahrens

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

Iteratives Typisierungsverfahren

Regel zur Berechnung neuer Annahmen:

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

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

Genau wie RSK1, aber in A 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

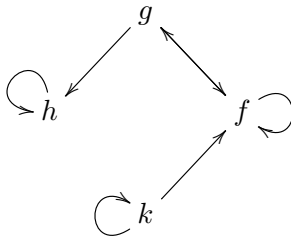
Typisierung der Gruppen entsprechend der $\preceq^+$ -Ordnung modulo $\simeq$.

Iteratives Typisierungsverfahren: Vorarbeiten (2)

Beispiel:

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

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



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

Iteratives Typisierungsverfahren: Der Algorithmus

Iterativer Typisierungsalgorithmus

Eingabe: Menge von verschränkt rekursiven Superkombinatoren

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

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

Iteratives Typisierungsverfahren: Der Algorithmus

Iterativer Typisierungsalgorithmus

Eingabe: Menge von verschränkt rekursiven Superkombinatoren

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

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

Iteratives Typisierungsverfahren: Der Algorithmus

Iterativer Typisierungsalgorithmus

Eingabe: Menge von verschränkt rekursiven Superkombinatoren

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

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

Iteratives Typisierungsverfahren: Der Algorithmus

Iterativer Typisierungsalgorithmus

Eingabe: Menge von verschränkt rekursiven Superkombinatoren

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

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

Iteratives Typisierungungsverfahren: Der Algorithmus

Iterativer Typisierungsalgorithmus

Eingabe: Menge von verschränkt rekursiven Superkombinatoren

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

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

Ausgabe Die allquantifizierten polymorphen Typen der SK_i

Iteratives Typisierungungsverfahren: Der Algorithmus

Iterativer Typisierungsalgorithmus

Eingabe: Menge von verschränkt rekursiven Superkombinatoren

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

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

Ausgabe Die allquantifizierten polymorphen Typen der SK_i

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:

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

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

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

wobei 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)

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

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

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

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

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

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

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

Beispiele: length (3)

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

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

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

$$\frac{}{\text{(RApP)} \frac{}{A'_0 \vdash (1 + \text{length } ys) :: \alpha_{10}, \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \dot{=} \text{Int} \rightarrow \alpha_{11}, \alpha_{13} \dot{=} \alpha_5 \rightarrow \alpha_{12}, \alpha_{11} \dot{=} \alpha_{12} \rightarrow \alpha_{10}\}}}$$

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

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

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

Beispiele: length (3)

Zusammengefasst:

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

wobei σ Lösung von

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

Die Unifikation ergibt als Unifikator

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

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

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

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

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

Iteratives Verfahren ist allgemeiner als Haskell

Beispiel

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

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

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

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

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

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

Nächste Iteration zeigt: A_1 ist konsistent.

Iteratives Verfahren ist allgemeiner als Haskell (2)

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

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

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

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

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

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

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

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

Iteratives Verfahren ist allgemeiner als Haskell (2)

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

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

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

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

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

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

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

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

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

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

```
Prelude> :t g
```

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

Grund: Wenn Typ vorhanden, führt Haskell keine Typinferenz durch, sondern verifiziert nur die Annahme. g wird im Rumpf wie bereits typisiert behandelt.

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

$$A_0 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_5 \rightarrow [\alpha_5]$$

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

Bsp.: Mehrere Iterationen sind nötig (2)

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

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

$$\begin{array}{c}
 \frac{
 \frac{
 \frac{
 \frac{
 \frac{
 A'_1 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset
 }{(AXK)}
 , \frac{
 A'_1 \vdash x :: \alpha_1, \emptyset
 }{(AXV)}
 }{(RAPP)}
 , \frac{
 \frac{
 \frac{
 A'_1 \vdash g :: \alpha_6 \rightarrow [\alpha_6], \emptyset
 }{(AXSK)}
 , \frac{
 \frac{
 A'_1 \vdash (g \text{ 'c'}) :: \alpha_7, \{\alpha_8 \rightarrow [\alpha_8] \dot{=} \text{Char} \rightarrow \alpha_7\}
 }{(RAPP)}
 }{(RAPP)}
 }{(RAPP)}
 }{(RAPP)}
 }{(RAPP)}
 }{(RAPP)}
 }{(RAPP)}
 }{(SKREK)}
 }{
 A'_1 \vdash \text{Cons } x (g (g \text{ 'c'})) :: \alpha_2, \{\alpha_8 \rightarrow [\alpha_8] \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6] \dot{=} \alpha_7 \rightarrow \alpha_4 \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}
 }
 }{
 A_1 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = [\text{Char}] \rightarrow [[\text{Char}]]
 }
 }{
 \text{wobei } \sigma = \{\alpha_1 \mapsto [\text{Char}], \alpha_2 \mapsto [[\text{Char}]], \alpha_3 \mapsto [[[\text{Char}]]], \alpha_4 \mapsto [[[\text{Char}]]], \alpha_5 \mapsto [\text{Char}], \alpha_6 \mapsto [\text{Char}], \alpha_7 \mapsto [\text{Char}], \alpha_8 \mapsto \text{Char}\}
 }
 }{
 \text{die Lösung von } \{\alpha_8 \rightarrow [\alpha_8] \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6] \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\} \text{ ist.}
 }$$

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

Bsp.: Mehrere Iterationen sind nötig (3)

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

Sei $A'_2 = A_2 \cup \{x :: \alpha_1\}$:

$$\begin{array}{c}
 \text{(AxSK)} \frac{A'_2 \vdash \text{g} :: [\text{Char}] \rightarrow [[\text{Char}]], \emptyset}{A'_2 \vdash \text{g} :: [\text{Char}] \rightarrow [[\text{Char}]], \emptyset}, \text{(AxV)} \frac{A'_2 \vdash x :: \alpha_1, \emptyset}{A'_2 \vdash x :: \alpha_1, \emptyset}, \text{(RApp)} \frac{A'_2 \vdash (\text{g } 'c') :: \alpha_7, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7\}}{A'_2 \vdash (\text{g } 'c') :: \alpha_7, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7\}}, \text{(RApp)} \frac{A'_2 \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3}{A'_2 \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3}, \text{(RApp)} \frac{A'_2 \vdash (\text{g } (\text{g } 'c')) :: \alpha_4, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7, [\text{Char}] \rightarrow [[\text{Char}]] \doteq \alpha_7 \rightarrow \alpha_2\}}{A'_2 \vdash (\text{g } (\text{g } 'c')) :: \alpha_4, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7, [\text{Char}] \rightarrow [[\text{Char}]] \doteq \alpha_7 \rightarrow \alpha_2\}}, \text{(SKRRed)} \frac{A'_2 \vdash \text{Cons } x (\text{g } (\text{g } 'c')) :: \alpha_2, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7, [\text{Char}] \rightarrow [[\text{Char}]] \doteq \alpha_7 \rightarrow \alpha_4 \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}}{A'_2 \vdash \text{Cons } x (\text{g } (\text{g } 'c')) :: \alpha_2, \{[\text{Char}] \rightarrow [[\text{Char}]] \doteq \text{Char} \rightarrow \alpha_7, [\text{Char}] \rightarrow [[\text{Char}]] \doteq \alpha_7 \rightarrow \alpha_4 \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}}
 \end{array}$$

$A_2 \vdash_T \text{g} :: \sigma(\alpha_1 \rightarrow \alpha_2)$

wobei σ die Lösung von

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

Unifikation:

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

g ist nicht typisierbar.

Daher gilt ...

Beobachtung

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

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

Nichtterminierung des iterativen Verfahrens

$$f = [g]$$

$$g = [f]$$

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

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

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

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

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

Nichtterminierung des iterativen Verfahrens (2)

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

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

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

Nichtterminierung des iterativen Verfahrens (3)

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

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

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

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

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

Nichtterminierung des iterativen Verfahrens (4)

Vermutung: Terminiert nicht

Beweis: (Induktion) betrachte den i . Schritt:

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

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

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

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

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

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

Daher ...

Beobachtung

Das iterative Typisierungsverfahren terminiert nicht immer.

Daher ...

Beobachtung

Das iterative Typisierungsverfahren terminiert nicht immer.

Es gilt sogar:

Satz

Die iterative Typisierung ist unentscheidbar.

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

Aufrufhierarchie

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

Type Safety (einer Programmiersprache)

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

- („Type Preservation“)

Die Typisierung bleibt unter Reduktion erhalten

Für einen Grundtyp τ :

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

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

D.h. Typen der Ausdrücke können allgemeiner werden.

- („Progress Lemma“):

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

Type Safety (2)

Lemma

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

Beweis: s direkt dynamisch ungetypt ist, gdw.:

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

Type Safety (3)

Lemma (Type Preservation)

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

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

Type Safety (4)

Aus den letzten beiden Lemmas folgt:

Satz

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

Lemma (Progress Lemma)

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

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

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

Type Safety (5)

Satz

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

Hindley-Milner Typisierung

Hindley-Milner Typisierung als Einschränkung der iterativen Typisierung

Roger Hindley; Robin Milner und Luis Damas haben beigetragen.

Erzwingen der Terminierung (der Typcheck-Iteration)

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

Hindley-Milner-Schritt: Typisiere $SK_1, \dots, SK_m$ auf einmal, mit der Annahme:

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

ohne Quantoren

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

Erzwingen der Terminierung (2)

für $i = 1, \dots, m$:

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

Als zusätzliche Regel muss im Typisierungsverfahren hinzugefügt werden:

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

Erzwingen der Terminierung (3)

Unterschied zum iterativen Schritt:

- Die Typen der zu typisierenden SKs werden nicht allquantifiziert.
(allquantifiziert sind die bekannten Typen von anderen SKs.)
- Daher sind während der Typisierung **keine Kopien** 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.
(allquantifiziert sind die bekannten Typen von anderen SKs.)
- 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 Hindley-Milner-Schritt terminiert das Verfahren sofort.

Das Hindley-Milner-Typisierungsverfahren

Hindley-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 Hindley-Milner-Typisierungsverfahren

Hindley-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 Hindley-Milner-Typisierungs-Verfahren.
Allerdings erweitert. . .

Das Hindley-Milner-Typisierungsverfahren, genauer

Hindley-Milner-Typisierungsverfahren:

$SK_1, \dots, SK_m$ sind alle SKs einer Äquivalenzklasse bzgl. $\simeq$ wobei alle kleineren (benutzten) SKs bereits getypt sind.

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

für $i = 1, \dots, m$:

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

wenn 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 Hindley-Milner-typisierbar

Das Hindley-Milner-Typisierungsverfahren (2)

Vereinfachung: Regel für einen rekursiven SK

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

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 Hindley-Milner-Typcheck

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

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

Beispiele: Viele Typvariablen

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

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

Die Anzahl der Typvariablen ist 2^6 .

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

Beispiele: map

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

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

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

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

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

(a) bis (e) folgt

Beispiele: map (2)

(a)
$$\frac{(\text{AxV})}{A' \vdash xs :: \alpha_2, \emptyset}$$

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

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

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

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

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

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

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

Beispiele: map (3)

(e)

$$\begin{array}{c}
 \frac{\frac{(AxK) \overline{A'' \vdash \text{Cons} :: \alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}], \emptyset}}{(RAPP)} \quad \frac{\frac{(AxV) \overline{A'' \vdash f :: \alpha_1, \emptyset}, (AxV) \overline{A'' \vdash y :: \alpha_3, \emptyset}}{(RAPP)} \quad \frac{(AxSK2) \overline{A'' \vdash \text{map} :: \beta, \emptyset}, (AxV) \overline{A'' \vdash f :: \alpha_1, \emptyset}}{(RAPP)} \quad \frac{(AxV) \overline{A'' \vdash ys :: \alpha_4, \emptyset}}{(RAPP)} \\
 \frac{\frac{(RAPP) \overline{A'' \vdash (\text{Cons } (f \ y)) :: \alpha_{11}, \{\alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \dot{=} \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}\}}{(RAPP)} \quad \frac{(RAPP) \overline{A'' \vdash (\text{map } f) :: \alpha_{12}, \{\beta \dot{=} \alpha_1 \rightarrow \alpha_{12}\}}, (RAPP) \overline{A'' \vdash (\text{map } f \ ys) :: \alpha_{13}, \{\beta \dot{=} \alpha_1 \rightarrow \alpha_{12}, \alpha_{12} \dot{=} \alpha_4 \rightarrow \alpha_{13}\}}}{(RAPP) \overline{A'' \vdash (\text{Cons } (f \ y) (\text{map } f \ ys)) :: \alpha_{14}, \{\alpha_{11} \dot{=} \alpha_{13} \rightarrow \alpha_{14}, \alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \dot{=} \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}, \beta \dot{=} \alpha_1 \rightarrow \alpha_{12}, \alpha_{12} \dot{=} \alpha_4 \rightarrow \alpha_{13}\}}}
 \end{array}$$

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

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

Beispiele: map (4)

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

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

Die Unifikation ergibt

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

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

Beispiele: erneute Betrachtung

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

Iteratives Verfahren liefert Fail nach mehreren Iteration.

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

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

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

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

Beispiele: erneute Betrachtung (2)

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

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

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

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

wobei σ die Lösung von

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

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

Iteratives Verfahren kann allgemeinere Typen liefern

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

Die Typen für die Konstruktoren sind

`Leer ::  $\forall a.$  Baum  $a$` und

`Knoten ::  $\forall a.$   $a \rightarrow$  Baum  $a \rightarrow$  Baum  $a \rightarrow$  Baum  $a$`

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

Hindley-Milner-Typcheck `g ::  $a \rightarrow a \rightarrow$  Baum Bool`

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

Grund (im Verfahren):

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

Haskell akzeptiert für g die allgemeinere Typannahme:

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

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

Hindley-Milner Typisierung und Type Safety

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

Hindley-Milner Typisierung und Type Safety (2)

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

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

ist Hindley-Milner-typisierbar.

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

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

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

Hindley-Milner Typisierung und Type Safety (2)

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

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

ist Hindley-Milner-typisierbar.

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

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

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

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

Hindley-Milner Typisierung und Type Safety (3)

Im Let-Kernsprache: Hindley-Milner-typisierbar:

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

NICHT Hindley-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

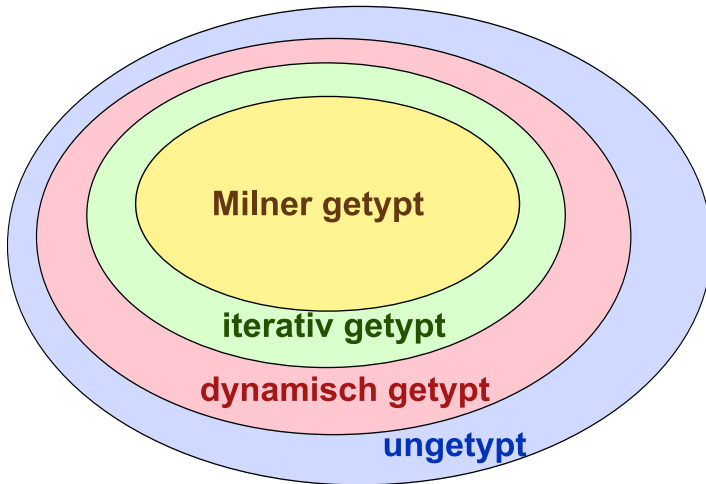
Nachher: alles wird auf einmal typisiert.

Hindley-Milner Typisierung und Type Safety(4)

Das Beispiel ist aber unkritisch, denn:

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

KFPTS+seq



Prädikativ / Imprädikativ

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

Prädikativ / Imprädikativ

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

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

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

Prädikativ / Imprädikativ

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

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

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

Mit imprädikativem Polymorphismus geht das:

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

Prädikativ / Imprädikativ

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

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

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

Mit imprädikativem Polymorphismus geht das:

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

Aber:

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

Typisierung unter Typklassen

Typisierung unter Typklassen

Typisierung unter Typklassen

Annahmen:

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

Beispiel

`genericLength :: Num b => [a] → b`

berechnet aus der Definition

Beschränkung kommt von der Addition `+`.

Erweiterung um Typklassen

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

Beispiel

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

Typklassenaxiome

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

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

Berechnungen auf Typklassenconstraints

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

Verfahren: Typklassenconstraints entscheiden

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

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

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

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

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

Typklassenconstraints: Beispiel

Gegeben als Axiome:

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

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

Typklassenconstraints: Beispiel

Gegeben als Axiome:

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

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

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

Typklassenconstraints: Beispiel

Gegeben als Axiome:

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

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

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

Typklassenconstraints: Beispiel

Gegeben als Axiome:

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

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

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

Typklassenconstraints: Beispiel

Gegeben als Axiome:

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

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

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

Was ist ein Baumautomat?

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

Ein Baum B wird folgendermaßen verarbeitet von einem Baumautomaten T :

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

Beispiel

Aussagenlogische Auswertung

.

Typklassenconstraints testen und Baumautomaten

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

Typklassen: Semantik bzgl. Grundtypen

Definition

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

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

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

Typklassen: Beispiele für Typisierung

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

Typklassen: Beispiele für Typisierung

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

Beispiel Typisierung

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

Typklassen: Beispiele für Typisierung

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

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

Beispiel Typisierung

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

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

Typklassen: Beispiele für Typisierung

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

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

Beispiel Typisierung

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

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

Es ergibt sich:

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

Typisierung unter Typklassen: Unifikation

Änderungen der Unifikation:

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

Typisierung unter Typklassen: Unifikation

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

Typisierung unter Typklassen: Beispiel

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

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

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

Typisierung unter Typklassen: Beispiel

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

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

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

$x :: \alpha$.

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

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

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

Typisierung unter Typklassen: Beispiel

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

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

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

$x :: \alpha$.

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

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

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

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

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

Typisierung unter Typklassen: Beispiel

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

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

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

$x :: \alpha$.

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

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

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

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

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

Implikation und Basisconstraints zeigen, dass das Constraint gilt!

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

Typisierung unter Typklassen: Beispiel

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

Typisierung unter Typklassen: Beispiel

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

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

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

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

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

Typisierung unter Typklassen: Beispiel

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

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

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

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

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

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

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

Typisierung unter Typklassen: Beispiel

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

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

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

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

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

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

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

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

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

Typisierung unter Typklassen: Beispiel

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

Typisierung; ergibt Gleichungen und Bedingungen:

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

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

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

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

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

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

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