GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Einführung in die Funktionale Programmierung:

Typisierung

Prof Dr. Manfred Schmidt-Schauß

WS 2022/23

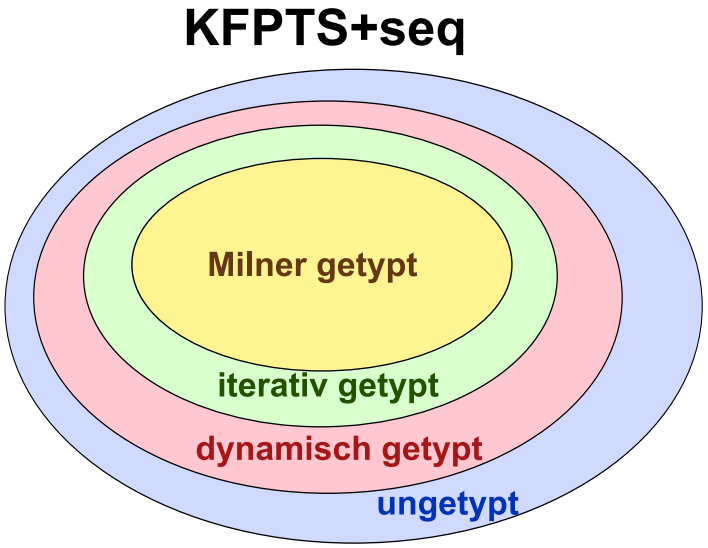
Stand der Folien: 19. Januar 2023

Übersicht

Ziele des Kapitels

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

Übersicht: Expression und Typen



Motivation

Warum ist ein Typsystem sinnvoll?

- Für ungetypte Programme können dynamische Typfehler auftreten
- Fehler zur Laufzeit sind Programmierfehler
- Starkes und statisches Typsystem
  $\Longrightarrow$ 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.

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 Haskells 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.

Funktioniert nicht gut, denn

Die dynamische Typisierung in KFPTS+seq ist unentscheidbar!

Unentscheidbarkeit der dynamischen Typisierung

Sei `tmEncode` eine KFPTS+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 KFPTS+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

$$
\begin{array}{ll}
s := & \texttt{if tmEncode } b\ e \\
& \quad \texttt{then case}_{\mathsf{Bool}}\ \texttt{Nil of } \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\} \\
& \quad \texttt{else case}_{\mathsf{Bool}}\ \texttt{Nil of } \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\}
\end{array}
$$

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:

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

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 $\alpha_1, \ldots, \alpha_n$
- Dann ist $\forall \alpha_1, \ldots, \alpha_n.\tau$ 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:
$\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$.
Formal: Erweiterung auf Typen: σ_E: Abbildung von Typen auf Typen

$$\begin{array}{rcl} \sigma_E(TV) & := & \sigma(TV), \text{ falls } \sigma \text{ die Variable } TV \text{ abbildet} \\ \sigma_E(TV) & := & TV, \text{ falls } \sigma \text{ die Variable } TV \text{ nicht abbildet} \\ \sigma_E(TC\ T_1\ \ldots\ T_n) & := & TC\ \sigma_E(T_1)\ \ldots\ \sigma_E(T_n) \\ \sigma_E(T_1 \to T_2) & := & \sigma_E(T_1) \to \sigma_E(T_2) \end{array}$$

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

Semantik eines polymorphen Typs

Grundtypen-Semantik für polymorphe Typen:

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

Entspricht der Vorstellung von schematischen Typen:

Ein polymorpher Typ ist ein
Schema für eine Menge von Grundtypen

Typregeln

Regel für Anwendung $(s\ t)$:

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

Problem: Man muss richtige Instanz raten, z.B.
`map` :: $(\mathtt{a} \to \mathtt{b}) \to [\mathtt{a}] \to [\mathtt{b}]$
`not` :: $\mathtt{Bool} \to \mathtt{Bool}$
Typisierung von `map not`: Vor Anwendung der Regel muss der Typ von `map` instanziiert werden mit

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

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:

$$\textsc{Decompose1}\ \frac{\Gamma \cup \{TC\ \tau_1\ \ldots\ \tau_n \doteq TC\ \tau_1'\ \ldots\ \tau_n'\}}{\Gamma \cup \{\tau_1 \doteq \tau_1', \ldots, \tau_n \doteq \tau_n'\}}$$

$$\textsc{Decompose2}\ \frac{\Gamma \cup \{\tau_1 \to \tau_2 \doteq \tau_1' \to \tau_2'\}}{\Gamma \cup \{\tau_1 \doteq \tau_1', \tau_2 \doteq \tau_2'\}}$$

Unifikationsalgorithmus: Schlussregeln (2)

Orientierung, Elimination:

$$\textsc{Orient}\ \frac{\Gamma \cup \{\tau_1 \doteq \alpha\}}{\Gamma \cup \{\alpha \doteq \tau_1\}}$$

wenn τ_1 keine Typvariable und α Typvariable

$$\textsc{Elim}\ \frac{\Gamma \cup \{\alpha \doteq \alpha\}}{\Gamma}$$

wobei α Typvariable

Unifikationsalgorithmus: Schlussregeln (3)

Einsetzung, Occurs-Check:

$$\textsc{Solve}\ \frac{\Gamma \cup \{\alpha \doteq \tau\}}{\Gamma[\tau/\alpha] \cup \{\alpha \doteq \tau\}}$$

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

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

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

Unifikationsalgorithmus: Abbruchregeln

Fail-Regeln:

$$\textsc{Fail1}\ \frac{\Gamma \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (TC_2\ \tau'_1\ \ldots\ \tau'_m)\}}{\mathsf{Fail}} \quad \text{wenn } TC_1 \neq TC_2$$

$$\textsc{Fail2}\ \frac{\Gamma \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (\tau'_1 \to \tau'_2)\}}{\mathsf{Fail}}$$

$$\textsc{Fail3}\ \frac{\Gamma \cup \{(\tau'_1 \to \tau'_2) \doteq (TC_1\ \tau_1\ \ldots\ \tau_n)\}}{\mathsf{Fail}}$$

Beispiele

Beispiel 1: $\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}$:

$$\textsc{Decompose2}\ \frac{\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}}{\{a \doteq \texttt{Bool}, b \doteq \texttt{Bool}\}}$$

Beispiel 2: $\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}$:

$$\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}$$

$$\textsc{Decompose2}\ \frac{\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}}{\{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\}}$$

$$\textsc{Decompose2}\ \frac{\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}}{\textsc{Orient}\ \dfrac{\{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\}}}$$

$$\textsc{Decompose2}\ \frac{\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}}{\textsc{Orient}\ \dfrac{\{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\}}}$$

Beispiele (2)

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

$$\textsc{OccursCheck}\ \frac{\textsc{Solve}\ \dfrac{\{a \doteq [b], b \doteq [a]\}}{\{a \doteq [[a]], b \doteq [a]\}}}{\mathsf{Fail}}$$

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

$$\textsc{Decompose2}\ \frac{\{a \to [b] \doteq a \to (c \to d)\}}{\textsc{Elim}\ \dfrac{\{a \doteq a, [b] \doteq c \to d\}}{\textsc{Fail2}\ \dfrac{\{[b] \doteq c \to d\}}{\mathsf{Fail}}}}$$

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
$$\{\alpha_1 \doteq \tau_1, \ldots, \alpha_n \doteq \tau_n\},$$
wobei α_i paarweise verschiedene Typvariablen sind und kein α_i in irgendeinem τ_j vorkommt. Der Unifikator kann dann abgelesen werden als $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$.
- Liefert der Algorithmus einen Unifikator, dann ist es ein allgemeinster 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

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 allgemeinster Unifikator für $\tau_1 \doteq \tau_2 \to \alpha$ ist
und α neue Typvariable ist.

Beispiel: `map not`

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

wenn σ allgemeinster Unifikator für
$(a \to b) \to [a] \to [b] \doteq (\texttt{Bool} \to \texttt{Bool}) \to \alpha$ ist
und α neue Typvariable ist.

Unifikation ergibt $\{a \mapsto \texttt{Bool}, b \mapsto \texttt{Bool}, \alpha \mapsto [\texttt{Bool}] \to [\texttt{Bool}]\}$

Daher: $\sigma(\alpha) = [\texttt{Bool}] \to [\texttt{Bool}]$

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 $\alpha \to \tau$.
- 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:

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

Woher erhalten wir τ_1?

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

Beispiel:

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

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

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

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

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

Typisierungsregeln für KFPTS+seq Ausdrücke (1)

Axiom für Variablen:

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

Axiom für Konstruktoren:

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

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:

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

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:

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

wobei α neue Typvariable

Regel für `seq`:

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

Typisierungsregeln für KFPTS+seq Ausdrücke (4)

Regel für Abstraktionen:

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

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 KFPTS+seq Ausdrücke (5)

Typisierung eines `case`: Prinzipien

$$\begin{pmatrix} \texttt{case}_{Typ}\ s\ \texttt{of}\ \{ \\ (c_1\ x_{1,1}\ \ldots\ x_{1,\text{ar}(c_1)}) \to t_1; \\ \ldots; \\ (c_m\ x_{m,1}\ \ldots\ x_{m,\text{ar}(c_m)}) \to t_m\} \end{pmatrix}$$

- 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 $t_1, \ldots, t_m$ 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`:**

$$(\text{RCase})\ \frac{\begin{array}{l} A \vdash s :: \tau, E \\ \text{für alle } i = 1, \ldots, m: \\ \quad A \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,\text{ar}(c_i)} :: \alpha_{i,\text{ar}(c_i)}\} \vdash (c_i\ x_{i,1}\ \ldots\ x_{i,\text{ar}(ci)}) :: \tau_i, E_i \\ \text{für alle } i = 1, \ldots, m: \\ \quad A \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,\text{ar}(c_i)} :: \alpha_{i,\text{ar}(c_i)}\} \vdash t_i :: \tau_i', E_i' \end{array}}{A \vdash \begin{pmatrix} \texttt{case}_{Typ}\ s\ \texttt{of}\ \{ \\ (c_1\ x_{1,1}\ \ldots\ x_{1,\text{ar}(c_1)}) \to t_1; \\ \ldots; \\ (c_m\ x_{m,1}\ \ldots\ x_{m,\text{ar}(c_m)}) \to t_m\} \end{pmatrix} :: \alpha, E'}$$

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 allgemeinster Unifikator von E, dann gilt $s :: \sigma(\tau)$.

Optimierung

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 KFPTS+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: $A_0 = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a], \texttt{True} :: \texttt{Bool}\}$
α: Variablen; τ: Typen und E Gleichungsmengen, die berechnet werden.

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

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

$$(\text{RApp})\ \frac{(\text{RApp})\ \dfrac{A_0 \vdash \texttt{Cons} :: \tau_3, E_3, \quad A_0 \vdash \texttt{True} :: \tau_4, E_4}{A_0 \vdash (\texttt{Cons True}) :: \alpha_2, \{\tau_3 \doteq \tau_4 \to \alpha_2\} \cup E_3 \cup E_4}, \quad (\text{AxK})\ \overline{A_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}}{A_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, \{\tau_3 \doteq \tau_4 \to \alpha_2\} \cup E_3 \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \to \alpha_4\}}$$

$$(\text{RApp})\ \frac{(\text{RApp})\ \dfrac{(\text{AxK})\ \overline{A_0 \vdash \texttt{Cons} :: \alpha_1 \to [\alpha_1] \to [\alpha_1], \emptyset}, \quad A_0 \vdash \texttt{True} :: \tau_4, E_4}{A_0 \vdash (\texttt{Cons True}) :: \alpha_2, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \tau_4 \to \alpha_2\} \cup E_4}, \quad (\text{AxK})\ \overline{A_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}}{A_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \tau_4 \to \alpha_2\} \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \to \alpha_4\}}$$

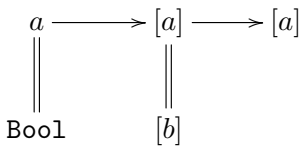
$$(\text{AxK})\ \frac{}{A_0 \vdash \texttt{Cons} :: \alpha_1 \to [\alpha_1] \to [\alpha_1], \emptyset}, \quad (\text{AxK})\ \frac{}{A_0 \vdash \texttt{True} :: \texttt{Bool}, \emptyset}$$

Beispiele: Typisierung von (Cons True Nil)

Vereinfachung für die Intuition und zum selbst Nachrechnen:
Typisierung von `Cons True Nil`

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

`Cons` :



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

Beispiele: Typisierung von $\lambda x.x$

Typisierung von $\lambda x.x$

Starte mit: Anfangsannahme: $A_0 = \emptyset$

$$(\text{RAbs})\ \frac{A_0 \cup \{x :: \alpha\} \vdash x :: \tau, E \overset{(\text{RAbs})}{\overset{(\text{AxV})}{}}}{A_0 \vdash (\lambda x.x) :: \alpha \to \tau, E}\ \frac{\overline{A_0 \cup \{x :: \alpha\} \vdash x :: \alpha, \emptyset}}{A_0 \vdash (\lambda x.x) :: \alpha \to \alpha, \emptyset}$$

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

Beispiele: Typisierung von Ω

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

Starte mit: Anfangsannahme: $A_0 = \emptyset$

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

$$(\text{RAbs})\ \frac{\{x :: \alpha_2\} \vdash (x\ x) :: \tau_6, E_1}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \tau_6, E_1}\ ,\ \ \emptyset \vdash (\lambda y.(y\ y)) :: \tau_2, E_2$$

$$(\text{RApp})\ \frac{\{x :: \alpha_2\} \vdash x :: \tau_3, E_3,\ \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,}{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_3, \{\tau_3 \doteq \tau_4 \to \alpha_3\} \cup E_3 \cup E_4}$$

$$(\text{RAbs})\ \frac{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_3, \{\tau_3 \doteq \tau_4 \to \alpha_3\} \cup E_3 \cup E_4}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \alpha_3, \{\tau_3 \doteq \tau_4 \to \alpha_3\} \cup E_3 \cup E_4}\ ,\ \emptyset \vdash (\lambda y.(y\ y)) :: \tau_2, E_2$$

$$(\text{RApp})\ \frac{}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, \{\tau_3 \doteq \tau_4 \to \alpha_3\} \cup E_3 \cup E_4 \cup E_2 \cup \{\alpha_2 \to \alpha_3 \doteq \tau_2 \to \alpha_1\}}$$

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

$$(\text{RApp})\ \frac{\overline{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}\ ,\ \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,}{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_3, \{\alpha_2 \doteq \tau_4 \to \alpha_3\} \cup E_4}$$

$$(\text{RAbs})\ \frac{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_3, \{\alpha_2 \doteq \tau_4 \to \alpha_3\} \cup E_4}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \alpha_3, \{\alpha_2 \doteq \tau_4 \to \alpha_3\} \cup E_4}\ ,\ \emptyset \vdash (\lambda y.(y\ y)) :: \tau_2, E_2$$

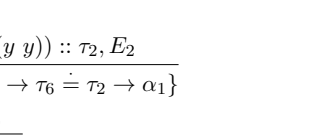
$$(\text{RApp})\ \frac{}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, \{\alpha_2 \doteq \tau_4 \to \alpha_3\} \cup E_4 \cup E_2 \cup \{\alpha_2 \to \alpha_3 \doteq \tau_2 \to \alpha_1\}}$$

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 $(x\ x)$:

x :



x

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

da das x in $\lambda x.(x\ x)$ monomorph typisiert werden muss.
Vorsicht: `(map map)` hat Typ $[a \to b] \to [[a] \to [b]]$

Beispiele: Typisierung eines Ausdrucks mit SKs (1)

Annahme:

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

Wir typisieren:

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

Als Anfangsannahme benutzen wir:

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

Beispiele: Typisierung eines Ausdrucks mit SKs (2)

Herleitungsbaum:

$$\dfrac{\dfrac{\overset{(\text{AxV})}{\overline{B_3}}, \overset{(\text{AxK})}{\overline{B_4}}, \overset{(\text{RApp})}{\dfrac{\overset{(\text{AxK})}{\overline{B_8}}, \overset{(\text{AxV})}{\overline{B_9}}}{B_6}}, \overset{(\text{AxV})}{\overline{B_7}}}{B_5}, \overset{(\text{AxK})}{\overline{B_{10}}}, \overset{(\text{RApp})}{\dfrac{\overset{(\text{AxSK})}{\overline{B_{14}}}, \overset{(\text{AxSK})}{\overline{B_{15}}}}{B_{12}}}, \overset{(\text{AxV})}{\overline{B_{13}}}}{B_{11}}}{B_2}\ (\text{RCase})}{B_1}\ (\text{RAbs})$$

Beschriftungen:

$$\begin{aligned}
B_1 = \quad & A_0 \vdash t :: \alpha_1 \to \alpha_{13},\\
& \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
& (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
& \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}\\
B_2 = \quad & A_0 \cup \{xs :: \alpha_1\} \vdash\\
& \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{map}\ \texttt{length}\ ys\} :: \alpha_{13},\\
& \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
& (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
& \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}
\end{aligned}$$

Beispiele: Typisierung eines Ausdrucks mit SKs (3)

Herleitungsbaum:

$$\dfrac{\dfrac{\overset{(\text{AxV})}{\overline{B_3}}, \overset{(\text{AxK})}{\overline{B_4}}, \overset{(\text{RApp})}{\dfrac{\overset{(\text{AxK})}{\overline{B_8}}, \overset{(\text{AxV})}{\overline{B_9}}}{B_6}}, \overset{(\text{AxV})}{\overline{B_7}}}{B_5}, \overset{(\text{AxK})}{\overline{B_{10}}}, \overset{(\text{RApp})}{\dfrac{\overset{(\text{AxSK})}{\overline{B_{14}}}, \overset{(\text{AxSK})}{\overline{B_{15}}}}{B_{12}}}, \overset{(\text{AxV})}{\overline{B_{13}}}}{B_{11}}}{B_2}\ (\text{RCase})}{B_1}\ (\text{RAbs})$$

Beschriftungen:

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

Beispiele: Typisierung eines Ausdrucks mit SKs (4)

Herleitungsbaum:

$$\dfrac{\dfrac{\overset{(\text{AxV})}{\overline{B_3}}, \overset{(\text{AxK})}{\overline{B_4}}, \overset{(\text{RApp})}{\dfrac{\overset{(\text{AxK})}{\overline{B_8}}, \overset{(\text{AxV})}{\overline{B_9}}}{B_6}}, \overset{(\text{AxV})}{\overline{B_7}}}{B_5}, \overset{(\text{AxK})}{\overline{B_{10}}}, \overset{(\text{RApp})}{\dfrac{\overset{(\text{AxSK})}{\overline{B_{14}}}, \overset{(\text{AxSK})}{\overline{B_{15}}}}{B_{12}}}, \overset{(\text{AxV})}{\overline{B_{13}}}}{B_{11}}}{B_2}\ (\text{RCase})}{B_1}\ (\text{RAbs})$$

Beschriftungen:

$$\begin{aligned}
B_{11} = \quad & A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map}\ \texttt{length})\ ys :: \alpha_{12},\\
& \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12}\}\\
B_{12} = \quad & A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map}\ \texttt{length}) :: \alpha_{11},\\
& \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}\}\\
B_{13} = \quad & A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset\\
B_{14} = \quad & A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{map} :: (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9], \emptyset\\
B_{15} = \quad & A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{length} :: [\alpha_{10}] \to \texttt{Int}, \emptyset
\end{aligned}$$

Beispiele: Typisierung eines Ausdrucks mit SKs (5)

Beschriftung unten:

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

Löse mit Unifikation:

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

Ergibt:

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

Damit erhält man $t :: \sigma(\alpha_1 \to \alpha_{13}) = [[\alpha_{10}]] \to [\texttt{Int}]$.
zur Erinnerung:
$t := \lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{map length}\ ys\}$

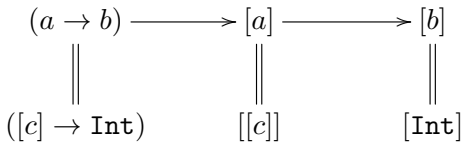
Vereinfachung Typisierung von (map length xs)

für die Intuition und selbst Nachrechnen:
Typisierung von map length xs

Starte mit:
Ann: $A_0 = \{\texttt{map} :: \forall a, b.(a \to b) \to [a] \to [b], \texttt{length} :: \forall a.[a] \to \texttt{Int}\}$

Typisiere map length xs

map :



length xs (Resultat)

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

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

Die Funktion const ist definiert als

```
const :: a -> b -> a
const x y = x
```

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

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

Anfangsannahme:
$A_0 = \{\texttt{const} :: \forall a, b.a \to b \to a, \texttt{True} :: \texttt{Bool}, \texttt{'A'} :: \texttt{Char}\}$.

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

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

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

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

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

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')
```

- 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**.
- KFPTS+seq hat kein `let`, aber Superkombinatoren, die wie (ein eingeschränkten 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.
- $SK_1, \ldots, SK_m$ sind verschränkt rekursiv, wenn $SK_i \preceq^+ SK_j$ für alle $i, j \in \{1, \ldots, m\}$

Typisierung von nicht-rekursiven Superkombinatoren

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

Typisierungsregel für (geschlossene) nicht-rekursive SK:

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

wenn σ Lösung von E,

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

und SK nicht rekursiv ist,

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

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

Beispiel: Typisierung von (.)

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

A_0 ist leer, da keine Konstruktoren oder SK vorkommen.

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

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

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

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

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

Typisierung von rekursiven Superkombinatoren

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

Idee des Iterativen Typisierungsverfahrens

- Gebe SK zunächst den allgemeinsten Typ (d.h. eine Typvariable) und typisiere den Rumpf unter Benutzung dieses Typs
- Man erhält anschließend einen neuen Typ für SK
- Mache mit neuem (quantifizierten) Typ im Rumpf weiter.
- Stoppe, wenn neuer Typ = alter Typ
- Dann hat man eine konsistente Typannahme gefunden; Vermutung: auch eine ausreichend allgemeine (allgemeinste?)

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

Iteratives Typisierungsverfahren

Regel zur Berechnung neuer Annahmen:

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

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

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

Iteratives Typisierungsverfahren: Vorarbeiten (1)

Wegen verschränkter Rekursion:

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

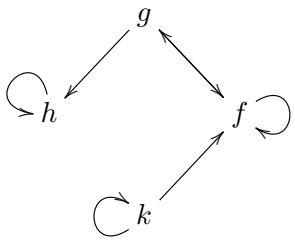
Typisierung der Gruppen entsprechend der $\preceq^+$-Ordnung modulo $\simeq$.

Iteratives Typisierungsverfahren: Vorarbeiten (2)

Beispiel:

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

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



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

Iteratives Typisierungsverfahren: Der Algorithmus

Iterativer Typisierungsalgorithmus

Eingabe: Menge von verschränkt rekursiven Superkombinatoren $SK_1, \ldots, SK_m$ wobei "kleinere" SK's schon typisiert; (keine freien Variablen)

1. Anfangsannahme A enthält Typen der Konstruktoren der bereits bekannten Superkombinatoren
2. $A_0 := A \cup \{SK_1 :: \forall\alpha_1.\alpha_1, \ldots, SK_m :: \forall\alpha_m.\alpha_m\}$ und $j = 0$.
3. Verwende für jeden Superkombinator SK_i (mit $i = 1, \ldots, m$) die Regel (SKR_{EK}) 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: $SK_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SK_m :: \forall\mathcal{X}_m.\tau_m$
   Setze $A_{j+1} := A \cup \{SK_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SK_m :: \forall\mathcal{X}_m.\tau_m\}$
5. Wenn $A_j \neq A_{j+1}$ (=: Gleichheit bis auf Umbenennung), dann gehe mit $j := j + 1$ zu Schritt (3). Anderenfalls, d.h. wenn $A_j = A_{j+1}$, war A_j konsistent; die Typen der SK_i sind entsprechend in A_j zu finden.
   Ausgabe Die allquantifizierten polymorphen Typen der SK_i

Sollte irgendwann ein `Fail` in der Unifikation auftreten, dann sind $SK_1, \ldots, SK_m$ nicht typisierbar.

Eigenschaften des Algorithmus

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

Beispiele: length (1)

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

Annahme:

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

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

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

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

wobei σ Lösung von

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

Beispiele: length (2)

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

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

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

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

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

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

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

Beispiele: length (3)

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

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

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

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

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

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

Beispiele: length (3)

Zusammengefasst:

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

wobei σ Lösung von

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

Die Unifikation ergibt als Unifikator

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

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

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

Da $A_0 \neq A_1$ muss man mit A_1 erneut iterieren.
2.Iteration: Ergibt den gleichen Typ, daher war A_1 konsistent.

Iteratives Verfahren ist allgemeiner als Haskell

Beispiel

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

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

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

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

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

Nächste Iteration zeigt: A_1 ist konsistent.

Iteratives Verfahren ist allgemeiner als Haskell (2)

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

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

<interactive>:1:13:
Couldn't match expected type '[t]' against inferred type 'Char'
      Expected type: Char -> [t]
      Inferred type: Char -> Char
    In the second argument of '(:)', namely '(g (g 'c'))'
    In the expression: 1 : (g (g 'c'))
```

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

```
let g::a -> [Int]; g x = 1:(g(g 'c'))
Prelude> :t g
g :: a -> [Int]
```

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

Bsp.: Mehrere Iterationen sind nötig (1)

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

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

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

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

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

Bsp.: Mehrere Iterationen sind nötig (2)

Da $A_0 \neq A_1$ muss eine weitere Iteration durchgeführt werden.
Sei $A_1' = A_1 \cup \{x :: \alpha_1\}$:

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

$$A_1 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \texttt{[Char]} \to \texttt{[[Char]]}$$

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

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

Bsp.: Mehrere Iterationen sind nötig (3)

Da $A_1 \neq A_2$ muss eine weitere Iteration durchgeführt werden:
Sei $A_2' = A_2 \cup \{x :: \alpha_1\}$:

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

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

wobei σ die Lösung von
$\{\texttt{[Char]} \to \texttt{[[Char]]} \doteq \texttt{Char} \to \alpha_7, \texttt{[Char]} \to \texttt{[[Char]]} \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ ist.

Unifikation:

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

g ist nicht typisierbar.

Daher gilt ...

Beobachtung

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

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

Nichtterminierung des iterativen Verfahrens

```
f = [g]
g = [f]
```

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

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

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

Nichtterminierung des iterativen Verfahrens (2)

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

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

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

Nichtterminierung des iterativen Verfahrens (3)

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

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

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

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

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

Nichtterminierung des iterativen Verfahrens (4)

Vermutung: Terminiert nicht

Beweis: (Induktion) betrachte den i. Schritt:

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

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

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

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

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

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

Daher ...

Beobachtung

Das iterative Typisierungsverfahren terminiert nicht immer.

Es gilt sogar:

Satz

Die iterative Typisierung ist unentscheidbar.

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

Aufrufhierarchie

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

Type Safety (einer Programmiersprache)

Man spricht von Type Safety wenn gilt:

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

Type Safety (2)

Lemma

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

Beweis: s direkt dynamisch ungetypt ist, gdw.:

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

Type Safety (3)

Lemma (Type Preservation)

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

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

Type Safety (4)

Aus den letzten beiden Lemmas folgt:

Satz

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

Lemma (Progress Lemma)

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

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

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

Type Safety (5)

Satz

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

Hindley-Milner Typisierung

Hindley-Milner Typisierung

als Einschränkung

der iterativen Typisierung

Roger Hindley; Robin Milner und Luis Damas haben beigetragen.

Erzwingen der Terminierung (der Typcheck-Iteration)

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

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

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

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

Erzwingen der Terminierung (2)

$$(\text{SKREKM})\ \frac{\begin{array}{c}\text{für } i=1,\ldots,m:\\ A_M \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i', E_i\end{array}}{A_M \vdash_T \text{für } i=1,\ldots,m\ SK_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i')}$$

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

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

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

Als zusätzliche Regel muss im Typisierungsverfahren hinzugefügt werden:

$$(\text{AXSK2})\ \frac{}{A \cup \{SK :: \tau\} \vdash SK :: \tau}$$

wenn τ nicht allquantifiziert ist

Erzwingen der Terminierung (3)

Unterschied zum iterativen Schritt:

- Die Typen der zu typisierenden SKs werden nicht allquantifiziert.
  (allquantifiziert sind die bekannten Typen von anderen SKs.)
- Daher sind während der Typisierung keine Kopien 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.

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

Das Hindley-Milner-Typisierungsverfahren, genauer

Hindley-Milner-Typisierungsverfahren:
$SK_1, \ldots, SK_m$ 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 $SK_1, \ldots, SK_m$ mit der Regel (MSKREK):

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

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

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

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

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

Das Hindley-Milner-Typisierungsverfahren (2)

Vereinfachung: Regel für einen rekursiven SK

$$(\text{MSKRek1})\ \frac{A \cup \{SK :: \beta, x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \vdash_T SK :: \sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

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

Eigenschaften des Hindley-Milner-Typcheck

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

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

Beispiele: Viele Typvariablen

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

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

Die Anzahl der Typvariablen ist 2^6.
Verallgemeinert man das Beispiel mit Parameter n, dann sind 2^n Typvariablen notwendig.

Beispiele: map

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

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

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

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

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

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

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

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

Beispiele: map (3)

(e)
$$\dfrac{\dfrac{\dfrac{}{A'' \vdash \texttt{Cons} :: \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}], \emptyset}\,(\text{AxK}) \quad \dfrac{\dfrac{}{A'' \vdash f :: \alpha_1, \emptyset}\,(\text{AxV}) \quad \dfrac{}{A'' \vdash y :: \alpha_3, \emptyset}\,(\text{AxV})}{A'' \vdash (f\ y) :: \alpha_{15}, \{\alpha_1 = \alpha_3 \to \alpha_{15}\}}\,(\text{RApp})}{A'' \vdash (\texttt{Cons}\ (f\ y)) :: \alpha_{11}, \{\alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\,(\text{RApp}) \quad \dfrac{\dfrac{}{A'' \vdash \texttt{map} :: \beta, \emptyset}\,(\text{AxSK2}) \quad \dfrac{}{A'' \vdash f :: \alpha_1, \emptyset}\,(\text{AxV})}{A'' \vdash (\texttt{map}\ f) :: \alpha_{12}, \{\beta \doteq \alpha_1 \to \alpha_{12}\}}\,(\text{RApp}) \quad \dfrac{}{A'' \vdash ys :: \alpha_4, \emptyset}\,(\text{AxV})}{A'' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \alpha_{14}, \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}\,(\text{RApp})$$
D.h. $\tau_5 = \alpha_{14}$ und
$E_5 = \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15},$
$\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}$

Beispiele: map (4)

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

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

Die Unifikation ergibt

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

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

Beispiele: erneute Betrachtung

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(\text{AxK}) \quad \dfrac{}{A' \vdash x :: \alpha, \emptyset}\,(\text{AxV})}{A' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3}\,(\text{RApp}) \quad \dfrac{\dfrac{}{A' \vdash \texttt{g} :: \beta, \emptyset}\,(\text{AxSK2}) \quad \dfrac{\dfrac{\dfrac{}{A' \vdash \texttt{g} :: \beta, \emptyset}\,(\text{AxSK2}) \quad \dfrac{}{A' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(\text{AxK})}{A' \vdash (\texttt{g 'c'}) :: \alpha_7, \{\beta \doteq \texttt{Char} \to \alpha_7\}}\,(\text{RApp})}{A' \vdash (\texttt{g}\ (\texttt{g 'c'})) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}\,(\text{RApp})}{A' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g 'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,(\text{RApp})}{A \vdash_T \texttt{g} :: \sigma(\alpha \to \alpha_2)}\,(\text{MSKRek})$$
wobei σ die Lösung von
$\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$ ist.

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

Beispiele: erneute Betrachtung (2)

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

Iteratives Verfahren liefert $g :: \forall\alpha.\alpha \to [\texttt{Int}]$
Hindley-Milner: Sei $A' = A \cup \{x :: \alpha, \texttt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(\text{AxK}) \quad \dfrac{}{A' \vdash 1 :: \texttt{Int}, \emptyset}\,(\text{AxK})}{A' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}\,(\text{RApp}) \quad \dfrac{\dfrac{}{A' \vdash \texttt{g} :: \beta, \emptyset}\,(\text{AxSK2}) \quad \dfrac{\dfrac{}{A' \vdash \texttt{g} :: \beta, \emptyset}\,(\text{AxSK2}) \quad \dfrac{}{A' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(\text{AxK})}{A' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\beta \doteq \texttt{Char} \to \alpha_7\}}\,(\text{RApp})}{A' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}\,(\text{RApp})}{A' \vdash \texttt{Cons}\ 1\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,(\text{SKRek})$$

$$A \vdash_T \texttt{g} :: \sigma(\alpha \to \alpha_2)$$

wobei σ die Lösung von

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

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

Iteratives Verfahren kann allgemeinere Typen liefern

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

Die Typen für die Konstruktoren sind
Leer $:: \forall a.$ Baum a und
Knoten $:: \forall a.\ a \to$ Baum a $\to$ Baum a $\to$ Baum a

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

Hindley-Milner-Typcheck g $:: a \to a \to$ Baum Bool
Iteratives Verfahren: g $:: a \to b \to$ Baum Bool
Grund (im Verfahren):
Iteratives Verfahren erlaubt Kopien des Typs für g, Hindley-Milner nicht.

Haskell akzeptiert für g die allgemeinere Typannahme:

```
g:: a -> b -> Baum Bool
g x y = Knoten True (g x y) (g y x)
```


Hindley-Milner Typisierung und Type Safety

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

Hindley-Milner Typisierung und Type Safety (2)

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

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

ist Hindley-Milner-typisierbar.

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

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

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

„Vermutlich“: Haskells operationale Semantik ist anders definiert

Hindley-Milner Typisierung und Type Safety (3)

Im Let-Kernsprache: Hindley-Milner-typisierbar:

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

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:

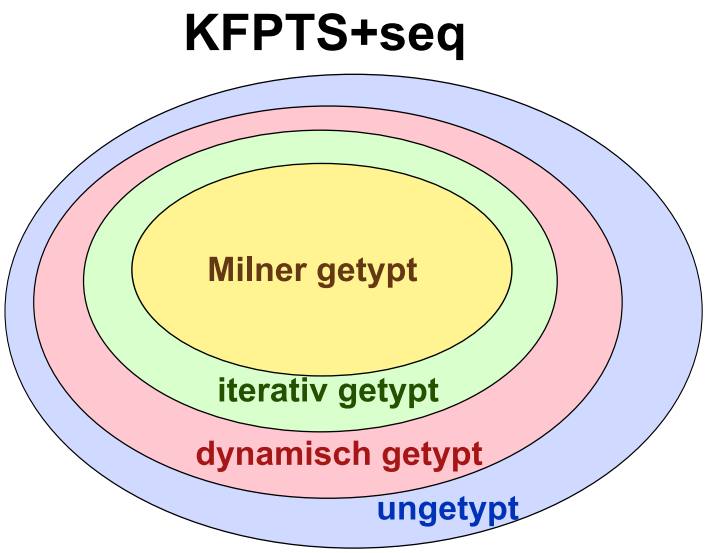
  Vorher: einmal kann allquantifiziert werden
  Nachher: alles wird auf einmal typisiert.

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

Übersicht



Prädikativ / Imprädikativ

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

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

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

Mit imprädikativem Polymorphismus geht das:

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

Aber:

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

Typisierung unter Typklassen

Typisierung unter Typklassen

Typisierung unter Typklassen

Annahmen:

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

Beispiel

`genericLength:: Num b => [a] → b`

berechnet aus der Definition
Beschränkung kommt von der Addition $+$.

Erweiterung um Typklassen

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

Beispiel

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

Typklassenaxiome

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

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

Berechnungen auf Typklassenconstraints

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

Verfahren: Typklassenconstraints entscheiden

Eingabe: Constraint-Menge $\mathcal{C} = \{Cl\ \tau\}$.
Vereinfache die Menge mit den zwei folgenden Schritten solange, bis die Menge leer ist und somit alle Constraints erfüllt.

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

Wenn die Constraint-Menge leer ist, dann ergibt sich `True`.
Wenn diese nicht komplett eliminiert werden kann, ergibt sich insgesamt `False`.

Typklassenconstraints: Beispiel

Gegeben als Axiome:

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

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

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

Was ist ein Baumautomat?

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

Ein Baum B wird folgendermaßen verarbeitet von einem Baumautomaten T:

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

Beispiel

Aussagenlogische Auswertung

. . . . .

Typklassenconstraints testen und Baumautomaten

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

Typklassen: Semantik bzgl. Grundtypen

Definition

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

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

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

Typklassen: Beispiele für Typisierung

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

Beispiel Typisierung

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

Typ von $+$ erzwingt: $a = \alpha_1 = \alpha_2$ und $\texttt{Num}\ a$.
Es ergibt sich:
$\lambda x.\lambda y.(x, y, x + y) :: \{\texttt{Num}\ a\}$ `=>` $a \to a \to (a, a, a)$

Typisierung unter Typklassen: Unifikation

Änderungen der Unifikation:

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

Typisierung unter Typklassen: Unifikation

- Analog sind die Änderungen bei den Typisierungsverfahren von Ausdrücken und Superkombinatoren.
- Dies gilt für das Hindley-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: `(x1,x2) + (y1,y2) = (x1+y1,x2+y2)`
Implikationsaxiom für Paare: $\{\texttt{Num}\ a, \texttt{Num}\ b\} \implies \texttt{Num}\ (a, b)$.

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

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

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

Typisierung unter Typklassen: Beispiel

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

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

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

Typisierung unter Typklassen: Beispiel

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

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