

Einführung in die Funktionale Programmierung:

Typisierung

funktionaler Programme

Prof Dr. Manfred Schmidt-Schauß

WS 2023/24

Stand der Folien: 9. Januar 2024

Ziele des Kapitels

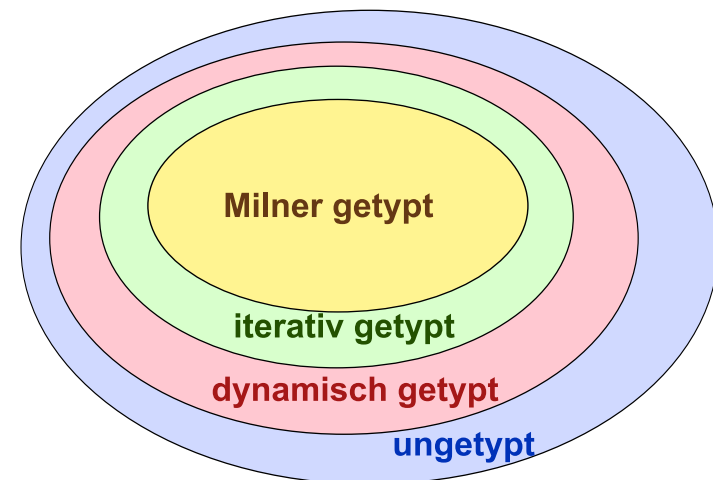
- Motivation: Warum typisieren?
- Typisierungsverfahren für Haskell bzw. KFPTS+seq für parametrisch polymorphe Typen ohne Superkombinatoren
 - Iteratives Typisierungsverfahren
 - Milnersches Typisierungsverfahren

Übersicht

- 1 Motivation
- 2 Typen: Sprechweisen, Notationen und Unifikation
- 3 Typisierungsverfahren

Übersicht: Ausdrücke und Typen

KFPTS+seq



Motivation

Warum ist ein Typsystem sinnvoll?

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

Motivation (3)

Es gibt Typsysteme, die diese Eigenschaften nicht erfüllen:

- Z.B. **Simply-typed Lambda-Calculus**: Getypte Sprache ist nicht mehr Turing-mächtig, da dieses Typsystem erzwingt, dass alle Programme **terminieren**
- Erweiterungen in Haskell's Typsystem:
Typisierung / Typinferenz ist unentscheidbar.
U.U. **terminiert der Compiler nicht!**
Folge: mehr Vorsicht/Anforderungen an den Programmierer.
- Typsysteme mit **dependent types** sind aktuell im Fokus der Forschung; und werden in Haskell-ähnlichen Programmiersprachen erprobt.
Diese sind komplexer als polymorphe Typisierung.

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**

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

- Dynamische Typisierung in KFPTS+seq ist **unentscheidbar!**
- Dynamische Typisierung gibt wenig Information über die Struktur des Programms

Unentscheidbarkeit der dynamischen Typisierung

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

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

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

(Haskell-Programm auf der Webseite, Archiv?)

Typen

Syntax von **polymorphen Typen**:

$$T ::= TV \mid TC \ T_1 \ \dots \ T_n \mid T_1 \rightarrow T_2$$

wobei TV Typvariable, TC Typkonstruktor

Sprechweisen:

- Ein **Basistyp** ist eine nullstelliger Typkonstruktor TC .
- 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.

Unentscheidbarkeit der dynamischen Typisierung (2)

Für eine TM-Beschreibung b und Eingabe e sei

```
s := if tmEncode b e
      then case_Bool Nil of {True -> True; False -> False}
      else case_Bool Nil of {True -> True; False -> False}
```

Es gilt:

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

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

Satz

Die dynamische Typisierung von $\text{KFPTS} + \text{seq}$ -Programmen ist unentscheidbar.

Typen (2)

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

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

Später: Allquantifizierte Typen dürfen kopiert und umbenannt werden,

Typen ohne Quantor dürfen nicht umbenannt werden!

Typsubstitutionen

Eine **Typsubstitution** ist eine Abbildung einer endlichen Menge von Typvariablen auf Typen, Schreibweise:

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

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

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

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

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

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

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

Typregeln

Regel für Anwendung ($s \ t$):

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

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

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

$$\text{not} :: \text{Bool} \rightarrow \text{Bool}$$

Typisierung von **map not**: Vor Anwendung der Regel muss der Typ

$(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ von **map** instanziiert werden mit

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

$$(\text{Bool} \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [b]$$

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

Semantik eines polymorphen Typs

Grundtypen-Semantik für polymorphe Typen:

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

Entspricht der Vorstellung von **schematischen** Typen:

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

Unifikationsproblem

Definition

Ein **Unifikationsproblem** auf Typen ist gegeben durch eine Menge Γ von Gleichungen der Form $\tau_1 = \tau_2$, 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 $\tau_1 = \tau_2$ 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 (2)

Orientierung, Elimination:

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

wenn τ_1 keine Typvariable und α Typvariable

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

wobei α Typvariable

Unifikationsalgorithmus: Schlussregeln (1)

Dekomposition:

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

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

Unifikationsalgorithmus: Schlussregeln (3)

Einsetzung, Occurs-Check:

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

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

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

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

Unifikationsalgorithmus: Abbruchregeln

Fail-Regeln:

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

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

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

Beispiele (2)

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

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

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

$$\text{DECOMPOSE2} \frac{\{a \rightarrow [b] \doteq a \rightarrow (c \rightarrow d)\}}{\{a \doteq a, [b] \doteq c \rightarrow d\}} \\ \text{ELIM} \frac{\{a \doteq a, [b] \doteq c \rightarrow d\}}{\{[b] \doteq c \rightarrow d\}} \\ \text{FAIL2} \frac{\{[b] \doteq c \rightarrow d\}}{\text{Fail}}$$

Beispiele

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

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

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

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

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

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

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

Eigenschaften des Unifikationsalgorithmus

- Der Algorithmus endet mit **Fail** gdw. es **keinen** Unifikator für die Eingabe gibt.
- Der Algorithmus endet erfolgreich gdw. es einen Unifikator für die Eingabe gibt. Das Gleichungssystem Γ ist dann von der Form

$$\{\alpha_1 \doteq \tau_1, \dots, \alpha_n \doteq \tau_n\},$$

wobei α_i paarweise verschiedene Typvariablen sind und kein α_i in irgendeinem τ_j vorkommt. Der Unifikator kann dann abgelesen werden als $\sigma = \{\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n\}$.

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

Eigenschaften des Unifikationsalgorithmus (2)

- Man braucht keine alternativen Regelanwendungen auszuprobieren! Der Algorithmus kann **deterministisch** implementiert werden.
- Der Algorithmus **terminiert** für jedes Unifikationsproblem auf Typen.
Ausgabe: Fail oder der allgemeinste Unifikator

Typisierungsverfahren

Wir betrachten nun die

polymorphe Typisierung

von KFPTSP+seq-Ausdrücken

Wir verschieben zunächst: Typisierung von Superkombinatoren

Typisierungsregeln:
arbeiten rekursiv auf der Struktur:
von Sub-Ausdrücken zum Ausdruck.

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.

Anwendungsregel mit Unifikation

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

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

Beispiel: map not

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

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

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

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

Anwendungsregel mit Unifikation

Beispiel rechnen: `map length`

`map length`

$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b], \text{length} :: [a'] \rightarrow \text{Nat}$

$(\text{map length}) :: \sigma(\alpha)$

wenn σ allgemeinsten Unifikator für die Gleichung

$(a \rightarrow b) \rightarrow [a] \rightarrow [b] \doteq ([a'] \rightarrow \text{Nat}) \rightarrow \alpha$ ist
und α neue Typvariable ist.

Dann ist der Unifikator: $\sigma = \{a \rightarrow [a']; b \rightarrow \text{Nat}\}$
und somit $\text{map length} : [[a']] \rightarrow \text{Nat}$

Typisierung mit Bindern (2)

Informelle Regel für die Abstraktion:

Typisierung von s unter der Annahme " x hat Typ τ_1 " ergibt $s :: \tau$
 $\lambda x.s :: \tau_1 \rightarrow \tau'$

Woher erhalten wir τ_1 ?

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

Beispiel:

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

Typisierung mit Bindern

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

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

Typisierung von Ausdrücken

Erweitertes Regelformat:

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

Bedeutung:

Gegeben eine Menge A von Typ-Annahmen.

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

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

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

Typisierung von Ausdrücken (2)

Herleitungsregeln schreiben wir in der Form

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

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

Andere Lesart:

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

Typisierungsregeln für KFPTS+seq Ausdrücke (1)

Axiom für Variablen:

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

Axiom für Konstruktoren:

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

wobei β_i neue Typvariablen sind

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

Typisierung von Ausdrücken (2)

Vereinfachung:

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

Typisierungsregeln für KFPTS+seq Ausdrücke (2)

Axiom für Superkombinatoren, deren Typ schon bekannt ist:

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

wobei β_i neue Typvariablen sind

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

Typisierungsregeln für KFPTS+seq Ausdrücke (3)

Regel für Anwendungen:

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

wobei α neue Typvariable

Regel für seq:

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

Typisierungsregeln für KFPTS+seq Ausdrücke (5)

Typisierung eines case: Prinzipien

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

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

Typisierungsregeln für KFPTS+seq Ausdrücke (4)

Regel für Abstraktionen:

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

wobei α eine neue Typvariable

In dieser Regel werden die Annahmen erweitert .

In E stehen die Bedingungen (Gleichungen) zu den Typvariablen.
 α wird normalerweise eingeschränkt.

Typisierungsregeln für KFPTS+seq Ausdrücke (6)

RCASE ist die Regel für case:

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

für alle $i = 1, \dots, m$:

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

für alle $i = 1, \dots, m$:

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

(RCASE)

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

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

Typisierungsalgorithmus für KFPTS+seq-Ausdrücke

Algorithmus:

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

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

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)

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

Beispiele: Typisierung von (Cons True Nil)

Typisierung von Cons True Nil

Starte mit:

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

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

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

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

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

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

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

Beispiele: Typisierung von (Cons True Nil)

Vereinfachung für die Intuition und zum selbst nachrechnen:

Typisierung von Cons True Nil

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

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

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

Beispiele: Typisierung von Ω

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

Starte mit: Anfangsannahme: $A_0 = \emptyset$

$$\begin{array}{c} \text{(RAPP)} \frac{\emptyset \vdash (\lambda x.(x x)) :: \tau_1, E_1, \quad \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha_1\}} \\ \text{(RABS)} \frac{\{x :: \alpha_2\} \vdash (x x) :: \tau_6, E_1}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \tau_6, E_1, \quad \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2} \\ \text{(RAPP)} \frac{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, E_1 \cup E_2 \cup \{\alpha_2 \rightarrow \tau_6 \doteq \tau_2 \rightarrow \alpha_1\}}{\{x :: \alpha_2\} \vdash x :: \tau_3, E_3, \quad \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,} \\ \text{(RAPP)} \frac{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4, \quad \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2} \\ \text{(RABS)} \frac{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4 \cup E_2 \cup \{\alpha_2 \rightarrow \alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_4 \cup E_2 \cup \{\alpha_2 \rightarrow \alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}} \\ \text{(AXV)} \frac{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset, \quad \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,}{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\alpha_2 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_4} \\ \text{(RABS)} \frac{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\alpha_2 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_4, \quad \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_4 \cup E_2 \cup \{\alpha_2 \rightarrow \alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}} \\ \text{(AXV)} \frac{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset, \quad \{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset,}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset, \quad \{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset,} \end{array}$$

Beispiele: Typisierung von $\lambda x.x$

Typisierung von $\lambda x.x$

Starte mit: Anfangsannahme: $A_0 = \emptyset$

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

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

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

Vereinfachung für die Intuition und zum selbst nachrechnen

Typisierung von $\lambda x.x$

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

Betrachte die Anwendung $(x x)$:

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

Erfordert: Lösung von $a = (a \rightarrow b)$ mit Typvariablen a, b ,
aber: die Gleichung hat keine Lösung !,
da das x in $\lambda x.(x x)$ monomorph typisiert werden muss.

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

Beispiele: Typisierung eines Ausdrucks mit SKs (5)

Beschriftung unten:

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

Löse mit Unifikation:

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

Ergibt:

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

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

zur Erinnerung:

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

Vereinfachung Typisierung von (map length xs)

für die Intuition und selbst Nachrechnen:

Typisierung von map length xs

Starte mit:

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

Typisiere map length xs

$$\begin{array}{ccccc}
 \text{map :} & (a \rightarrow b) & \longrightarrow & [a] & \longrightarrow & [b] \\
 & \parallel & & \parallel & & \parallel \\
 & ([c] \rightarrow \text{Int}) & & [[c]] & & [\text{Int}]
 \end{array}$$

length xs (Resultat)

Ergibt: $a = [c], b = \text{Int}$

$$(\text{map length } xs) :: [\text{Int}]$$

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

Die Funktion const ist definiert als

$$\begin{aligned}
 \text{const} & :: a \rightarrow b \rightarrow a \\
 \text{const } x \ y & = x
 \end{aligned}$$

Typisierung von $\lambda x. \text{const } (x \text{ True}) (x \text{ 'A'})$ Zum Beispiel: nach Einsetzen von $x = \text{Id}$ wäre der Rumpf getypt.

Anfangsannahme:

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

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

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

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

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

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

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

In Haskell:

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

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

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

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

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

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

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

- 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