

# Funktionale Programmierung

## Sommersemester 2025

Prof. Dr. Jakob Rehof

M.Sc. Felix Laarmann

TU Dortmund

LS XIV Software Engineering

# Diese Vorlesung:

- Grundprinzipien des Typsystems, Typcheck und Typinferenz

## Bestandteile

- Typen  $\sigma, \tau, \dots \in \mathbb{T}$

$$\sigma, \tau ::= a \mid A \mid \sigma \rightarrow \tau$$

mit

- Typvariablen  $a, b, \dots \in \mathbb{V}$
- Typkonstanten  $A, B, \dots \in \mathbb{C}$

- Typumgebungen (assumptions/contexts)

$$A = \{x_1 : \tau_1, \dots, x_n : \tau_n\} \text{ wobei } i \neq j \Rightarrow x_i \neq x_j$$

- Typaussagen (judgements)

$$A \vdash M : \tau$$

- Typregeln

$$\frac{A \vdash M : \sigma}{A' \vdash M' : \tau} (\text{r})$$

## Funktionsanwendung

- Bei einer *Funktionsanwendung*  $f\ e_1 \dots e_n$  müssen folgende Bedingungen erfüllt sein:
  - $f :: \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$  für irgendwelche Typen  $\sigma_1, \dots, \sigma_n, \tau$
  - $e_i :: \sigma_i, i = 1 \dots n$

Wenn dies der Fall ist, dann kann die Anwendung  $f\ e_1 \dots e_n$  mit dem Typ  $\tau$  getypt werden.

## Funktionsdefinition

- Bei einer *Funktionsdefinition*  $f \ x_1 \dots x_n = e$  kann die Funktion  $f$  mit einem Typ  $\sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$  getypt werden, wenn folgende Bedingung erfüllt ist:
  - Der Rumpf  $e$  kann mit dem Typ  $\tau$  getypt werden in einer *Typumgebung*, in welcher die Parameter  $x_i$  mit den Typen (jeweils)  $\sigma_i$  getypt sind. Diese Bedingung könnten wir auch formaler so ausdrücken:

$$\{x_1 :: \sigma_1, \dots, x_n :: \sigma_n\} \vdash e :: \tau$$

wobei die Relation  $\vdash$  die Typisierbarkeitsrelation  $A \vdash e :: \tau$  ist, die besagt, daß der Ausdruck  $e$  mit dem Typ  $\tau$  getypt werden kann, unter der Voraussetzung der in  $A$  gegebenen Typisierungen von Parametern.

- Bei explizit gegebenen Typdeklarationen, wie z.B.  $f :: a \rightarrow b \rightarrow a$ , müssen die oben genannten Bedingungen mit diesen Deklarationen übereinstimmen.

## Formalisierung

$$\frac{}{A \cup \{x :: \tau\} \vdash x :: \tau} (\text{par})$$

$$\frac{\begin{array}{c} A \vdash f :: \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau \\ A \vdash e_i :: \sigma_i \quad (i = 1 \dots n) \end{array}}{A \vdash f \ e_1 \dots e_n :: \tau} (\text{app})$$

$$\frac{A \cup \{x_1 :: \sigma_1, \dots, x_n :: \sigma_n\} \vdash e :: \tau}{\begin{array}{c} f :: \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau \\ \text{where } f \ x_1 \dots x_n = e \end{array}} (\text{def})$$

## Formalisierung - noch besser

$$\frac{}{A \cup \{x :: \tau\} \vdash x :: \tau} (\text{par})$$

$$\frac{\begin{array}{c} A \vdash e :: \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau \\ A \vdash e_i :: \sigma_i \quad (i = 1 \dots n) \end{array}}{A \vdash e \ e_1 \dots e_n :: \tau} (\text{app})$$

$$\frac{A \cup \{x_1 :: \sigma_1, \dots, x_n :: \sigma_n\} \vdash e :: \tau}{\begin{array}{c} f :: \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau \\ \text{where } f \ x_1 \dots x_n = e \end{array}} (\text{def})$$

# Typableitung

$$\frac{\overline{\{x :: a\} \vdash x :: a}}{\emptyset \vdash \begin{array}{c} \text{id} :: a \rightarrow a \\ \text{where } \text{id } x = x \end{array}} \text{(def)}$$

## Formalisierung - noch besser

Wir können uns vorstellen, daß diese Typregeln relativ zu einer (implizit gegebenen) Systemumgebung wie `Prelude` verwendet werden. So wenn wir schreiben

$$A \vdash e :: \tau$$

dann meinen wir in diesem Fall wirklich

$$\text{Prelude} \cup A \vdash e :: \tau$$

Diese Betrachtung können wir für beliebige Module `M` verwenden. Also würden wir in solchen Fällen eigentlich

$$M \cup A \vdash e :: \tau$$

betrachten.

Damit haben wir solche nützliche Dinge zur Verfügung wie z.B.

`'a'...` :: Char

`0...` :: Int

`True` :: Bool

`False` :: Bool

`( , )` ::  $a \rightarrow b \rightarrow (a,b)$

`fst` ::  $(a,b) \rightarrow a$

`snd` ::  $(a,b) \rightarrow b$

`[]` ::  $[a]$

`(:)` ::  $a \rightarrow [a] \rightarrow [a]$

`head` ::  $[a] \rightarrow a$

`tail` ::  $[a] \rightarrow [a]$

...

...

# Typableitung

$$\frac{\frac{\frac{\frac{\{x :: \text{Int}\} \vdash x :: \text{Int}}{(\text{var})}}{\emptyset \vdash \text{id} :: \text{Int} \rightarrow \text{Int}} (\text{def})}{\text{where id } x = x}}{\emptyset \vdash \text{id } 0 :: \text{Int}} (\text{app}) \quad \emptyset \vdash 0 :: \text{Int}$$

Noch eine regel bietet sich an:

$$\frac{A \vdash e :: \text{Bool} \quad A \vdash e_1 :: \tau \quad A \vdash e_2 :: \tau}{A \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 :: \tau} (\text{if})$$

# Simple typed $\lambda$ -calculus

## Bestandteile

- Typen  $\sigma, \tau, \dots \in \mathbb{T}$

$$\sigma, \tau ::= a \mid A \mid \sigma \rightarrow \tau$$

mit

- Typvariablen  $a, b, \dots \in \mathbb{V}$
- Typkonstanten  $A, B, \dots \in \mathbb{C}$

- Typumgebungen (assumptions/contexts)

$$A = \{x_1 : \tau_1, \dots, x_n : \tau_n\} \text{ wobei } i \neq j \Rightarrow x_i \neq x_j$$

- Typaussagen (judgements)

$$A \vdash M : \tau$$

- Typregeln

$$\frac{A \vdash M : \sigma}{A' \vdash M' : \tau} (\text{r})$$

## Simple typed $\lambda$ -calculus (implicit system)

$$\frac{}{A \cup \{x : \tau\} \vdash x : \tau} (\text{var})$$

$$\frac{A \vdash M : \sigma \rightarrow \tau \quad A \vdash N : \sigma}{A \vdash (M \ N) : \tau} (\text{app})$$

$$\frac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash \lambda x. M : \sigma \rightarrow \tau} (\text{abs})$$

## Simple typed $\lambda$ -calculus (explicit system)

$$\frac{}{A \cup \{x : \tau\} \vdash^* x : \tau} (\text{var})$$

$$\frac{A \vdash^* M : \sigma \rightarrow \tau \quad A \vdash^* N : \sigma}{A \vdash^* (M \ N) : \tau} (\text{app})$$

$$\frac{A \cup \{x : \sigma\} \vdash^* M : \tau}{A \vdash^* \lambda x : \sigma. M : \sigma \rightarrow \tau} (\text{abs})$$

## Substitutionseigenschaften

**Lemma (Substitution on terms).**

$$(A \cup \{x : \sigma\} \vdash M : \tau) \wedge (A \vdash N : \sigma) \Rightarrow A \vdash M[x := N] : \tau$$

Dieses Lemma ist das Prinzip der *Modularität* des Typsystems.

**Lemma (Substitution on types).**

$$A \vdash M : \tau \Rightarrow A[\alpha := \sigma] \vdash M : \tau[\alpha := \sigma]$$

Dieses Lemma ist das Prinzip des *Polymorphismus* des Typsystems.

## Polymorphismus

Es folgt unmittelbar aus dem Substitutionslemma auf Typen, dass jeder Ausdruck, der einen Typ mit mindestens einer Typvariable hat, unendlich viele *Typinstanzen* hat. Zum Beispiel haben wir

$$\emptyset \vdash \lambda x.x : \tau \rightarrow \tau, \text{ für alle } \tau$$

weil es unendlich viele entsprechende Ableitungen gibt. Für jeden Typ  $\tau$  gibt es ja die Ableitung

$$\frac{\overline{\{x : \tau\} \vdash x : \tau} \text{ (var)}}{\emptyset \vdash \lambda x.x : \tau \rightarrow \tau} \text{ (abs)}$$

Alle diese Typinstanzen gehen hervor als Substitutionsinstanzen von der Typisierung

$$\frac{\overline{\{x : \alpha\} \vdash x : \alpha} \text{ (var)}}{\emptyset \vdash \lambda x.x : \alpha \rightarrow \alpha} \text{ (abs)}$$

durch die Substitution (jeweils)  $[\alpha := \tau]$ .

Die Substitutionsprinzipien gelten grundsätzlich auch für Haskell.

Aus diesem Grund hat es Sinn, die Typen in Haskell als implizit *pränex universell quantorisiert* aufzufassen. Das heißt, wenn wir eine Typisierung haben

$$e :: \tau$$

dann können wir dieses auch so auffassen:

$$e :: \forall a_1 \dots a_n. \tau$$

wobei die  $a_1, \dots, a_n$  alle Typvariablen in  $\tau$  sind (die nicht in der Typumgebung vorkommen). Durch *Instanziierung* kann man für alle  $\sigma_1, \dots, \sigma_n$  ableiten:

$$e :: \tau[a_1 := \sigma_1] \dots [a_n := \sigma_n]$$

Die implizite pränexe Allquantorisierung wird häufig in der Mathematik benutzt. Zum Beispiel ist diese Formulierung der Kommutativität von  $+$

$$x + y = y + x$$

eine Verkürzung der vollständigen Form

$$\forall xy. x + y = y + x$$

# Prinzipale Typisierung

**Lemma (Prinzipalität).** Für jeden typisierbaren Term  $M$  existiert ein Prinzipiales Paar  $(A_p, \tau_p)$  mit folgenden Eigenschaften:

1.  $A_p \vdash M : \tau_p$
2. Für jedes Paar  $(A, \tau)$  mit  $A \vdash M : \tau$  existiert eine Substitution  $S$  mit  $S(\tau_p) = \tau$  und  $S(A_p) = A$

Hier  $S(A) = \{x : S(\sigma) \mid x : \sigma \in A\}$ . Der Typ  $\tau_p$  wird *prinzipaler Typ* von  $M$  genannt.

Mit anderen Worten: Jeder typisierbare Term hat eine *generellste* Typisierung, von welcher alle anderen Typisierungen durch Instanziierung hervorgehen. Der prinzipiale Typ eines Terms ist *eindeutig* (bis auf Umbenennung von Variablen).

# Entscheidungsprobleme

*Typinferenz (type inference)*

- Gegeben  $M$ :
  - existieren  $A$  und  $\tau$  mit  $A \vdash M : \tau$  ?

*Typcheck (type checking)*

- Gegeben  $A, M, \tau$ :
  - ist es der Fall, daß  $A \vdash M : \tau$  ?

GHCI kann beides.

## Typinferenz im einfach getypten Lambda-Kalkül

$$\frac{}{A \cup \{x : \tau\} \vdash x : \tau} (\text{var})$$

$$\frac{A \vdash M : \sigma \rightarrow \tau \quad A \vdash N : \sigma}{A \vdash (M \ N) : \tau} (\text{app})$$

$$\frac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash \lambda x. M : \sigma \rightarrow \tau} (\text{abs})$$

## Beispiel Typinferenz

Man soll den Typ (d.h. den *prinzipalen Typ*) vom Term  $\lambda f.\lambda x.x$  inferieren und den Ableitungsbaum konstruieren.

$$\frac{}{\{f : a, x : b\} \vdash x : b} \text{(var)}$$
$$\frac{}{\{f : a\} \vdash \lambda x. x : b \rightarrow b} \text{(abs)}$$
$$\frac{}{\emptyset \vdash \lambda f.\lambda x. x : a \rightarrow (b \rightarrow b)} \text{(abs)}$$

## Beispiel Typinferenz

Man soll den Typ (d.h. den *prinzipalen Typ*) vom Term  $\lambda f.\lambda x.(f\ x)$  inferieren und den Ableitungsbaum konstruieren.

$$\frac{\frac{\frac{\frac{\{f : a \rightarrow b, x : a\} \vdash f : a \rightarrow b}{(\text{var})}}{\{f : a \rightarrow b, x : a\} \vdash x : a}{(\text{var})}}{\{f : a \rightarrow b, x : a\} \vdash (f\ x) : b}{(\text{app})}}{\frac{\frac{\{f : a \rightarrow b\} \vdash \lambda x.(f\ x) : a \rightarrow b}{(\text{abs})}}{\emptyset \vdash \lambda f.\lambda x.(f\ x) : (a \rightarrow b) \rightarrow (a \rightarrow b)}{(\text{abs})}}{(\text{abs})}}$$

## Beispiel Typinferenz

Man soll den Typ (d.h. den *prinzipalen Typ*) vom Term  $\lambda f.\lambda x.(f (f x))$  inferieren und den Ableitungsbaum konstruieren.

$$\frac{}{\{f : a \rightarrow a, x : a\} \vdash f : a \rightarrow a} \text{(var)} \quad \frac{}{\{f : a \rightarrow a, x : a\} \vdash x : a} \text{(var)}$$

$$\frac{}{\{f : a \rightarrow a, x : a\} \vdash (f x) : a} \text{(app)}$$

$$\frac{\{f : a \rightarrow a, x : a\} \vdash f : a \rightarrow a}{\{f : a \rightarrow a, x : a\} \vdash (f x) : a} \text{(app)}$$

$$\frac{\{f : a \rightarrow a, x : a\} \vdash (f (f x)) : a}{\{f : a \rightarrow a\} \vdash \lambda x.(f (f x)) : a \rightarrow a} \text{(abs)}$$

$$\frac{\{f : a \rightarrow a\} \vdash \lambda x.(f (f x)) : a \rightarrow a}{\emptyset \vdash \lambda f.\lambda x.(f (f x)) : (a \rightarrow a) \rightarrow (a \rightarrow a)} \text{(abs)}$$

## Beispiel Typinferenz

Man soll den Typ (d.h. den *prinzipalen Typ*) vom Term

$$\lambda y.(x (\lambda z.(y (x y))))$$

in der Typumgebung

$$\{x : (a \rightarrow b) \rightarrow a\}$$

inferieren und den Ableitungsbaum konstruieren.

... das machen wir am besten am Tafel!

# Unifikation in einer Termalgebra

Termalgebra  $T_\Sigma(X)$ :

- $\Sigma$  eine Menge von Funktionssymbolen
- $X$  eine Menge von Variablen
- Terme  $t ::= x \mid f(t_1, \dots, t_m), x \in X, f \in \Sigma$

Ein Unifikationsproblem ist eine endliche Menge von formalen Gleichheiten (*Constraints*)

$$\mathcal{C} = \{x_i \doteq t_i \mid i = 1 \dots n\}$$

wobei  $t_i \in T_\Sigma(X)$  und  $x_i \in X$ .

Eine Lösung ist eine Substitution  $S = [x_1 := t_1, \dots, x_k := t_k]$  für die Variablen  $x_1, \dots, x_k$  in  $\mathcal{C}$  so daß  $S(x_i) = S(t_i)$ ,  $i = 1 \dots n$ .

Beispiel:

$$\mathcal{C} = \{f(x, y) \doteq f(z, g), z \doteq h(w)\}$$

Eine Lösung ist  $S = [x := h(w), y := g, z := h(w)]$

# Reduktion der Typisierbarkeit zu Unifikation

Wir assoziieren mit einem Term  $M$  eine Typvariable  $\alpha_M$  und eine Constraintmenge (Unifikationsconstraints)  $\mathcal{C}[M]$ :

$$\frac{}{A \cup \{x : \tau\} \vdash x : \tau} (\text{var}) \quad \mathcal{C}[x] = \emptyset \\ x : \alpha_x$$

$$\frac{A \vdash M : \sigma \rightarrow \tau \quad A \vdash N : \sigma}{A \vdash (M \ N) : \tau} (\text{app}) \quad \begin{aligned} \mathcal{C}[(MN)] &= \mathcal{C}[M] \cup \mathcal{C}[N] \cup \{\alpha_M \doteq \alpha_N \rightarrow \alpha_{(MN)}\} \\ (MN) &: \alpha_{(MN)} \end{aligned}$$

$$\frac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash \lambda x. M : \sigma \rightarrow \tau} (\text{abs}) \quad \begin{aligned} \mathcal{C}[\lambda x : M] &= \mathcal{C}[M] \cup \{\alpha_{\lambda x. M} \doteq \alpha_x \rightarrow \alpha_M\} \\ \lambda x. M &: \alpha_{\lambda x. M} \end{aligned}$$

# Reduktion zu Unifikation

$$\frac{A \vdash e :: \text{Bool} \quad A \vdash e_1 :: \tau \quad A \vdash e_2 :: \tau}{A \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 :: \tau} (\text{if})$$

$$\begin{aligned}\mathcal{C}[\text{if } \dots] &= \mathcal{C}[e_1] \cup \mathcal{C}[e_2] \cup \{\alpha_e \doteq \text{Bool}, \alpha_{e_1} \doteq \alpha_{e_2}, \alpha_{e_1} \doteq \alpha_{\text{if } \dots}\} \\ \text{if } e \text{ then } e_1 \text{ else } e_2 &: \alpha_{\text{if } \dots}\end{aligned}$$

# Reduktion zu Unifikation

$$\begin{aligned}\mathcal{C}[\text{True}] &= \{\alpha_{\text{True}} \doteq \text{Bool}\} \\ \mathcal{C}[0] &= \{\alpha_0 \doteq \alpha\}, \alpha \in \text{Num} \\ \mathcal{C}[(+)] &= \{\alpha_{(+)} \doteq \alpha \rightarrow \alpha \rightarrow \alpha\}, \alpha \in \text{Num} \\ \mathcal{C}[\text{fst}] &= \{\alpha_{\text{fst}} \doteq (\alpha, \beta) \rightarrow \alpha\} \\ \mathcal{C}[\text{head}] &= \{\alpha_{\text{head}} \doteq [\alpha] \rightarrow \alpha\} \\ &\dots\end{aligned}$$

## Reduktion zu Unifikation

**Lemma (Typinferenz).** Ein Term  $M$  ist typisierbar genau dann, wenn  $\mathcal{C}[M]$  unter Unifikation lösbar ist, und die generellste Lösung  $S$  (most general unifier) zum System  $\mathcal{C}[M]$  hat die Eigenschaft, dass das Paar

$$(\{x : S(\alpha_x) \mid x \in FV(M)\}, S(\alpha_M))$$

Prinzipal ist. Die generellste Lösung  $S$  ist eindeutig modulo Umbenennung von Variablen (renaming).

# Unifikationsalgorithmus (Robinson)

[https://en.wikipedia.org/wiki/Unification\\_\(computer\\_science\)#A\\_unification\\_algorithm](https://en.wikipedia.org/wiki/Unification_(computer_science)#A_unification_algorithm)

## Normalisierung von Constraints

$\mathcal{C} \cup \{t \doteq t\}$	$\Rightarrow \mathcal{C}$	<i>delete</i>
$\mathcal{C} \cup \{f(s_0, \dots, s_k) \doteq f(t_0, \dots, t_k)\}$	$\Rightarrow \mathcal{C} \cup \{s_0 \doteq t_0, \dots, s_k \doteq t_k\}$	<i>decompose</i>
$\mathcal{C} \cup \{f(s_0, \dots, s_k) \doteq g(t_0, \dots, t_m)\}$	$\Rightarrow \perp$	$f \neq g \vee k \neq m$ <i>conflict</i>
$\mathcal{C} \cup \{f(s_0, \dots, s_k) \doteq x\}$	$\Rightarrow \mathcal{C} \cup \{x \doteq f(s_0, \dots, s_k)\}$	<i>swap</i>
$\mathcal{C} \cup \{x \doteq t\}$	$\Rightarrow \mathcal{C}[x := t] \cup \{x \doteq t\}$	$x \notin \text{vars}(t) \wedge x \in \text{vars}(\mathcal{C})$ <i>eliminate</i>
$\mathcal{C} \cup \{x \doteq f(s_0, \dots, s_k)\}$	$\Rightarrow \perp$	$x \in \text{vars}(f(s_0, \dots, s_k))$ <i>check</i>

- Der Normalisierungsalgorithmus terminiert
- Wenn  $\mathcal{C} \Rightarrow \perp$ , dann existiert keine Lösung
- Wenn  $\mathcal{C} \not\Rightarrow \perp$ , dann sind alle Constraints nach Normalisierung in der Form  $x_i \doteq t_i$ , und  $S = [x_1 := t_1, \dots, x_n := t_n]$  ist eine Lösung. Diese ist die generellste Lösung in dem Sinne, daß für jede andere Lösung  $S'$  gilt  $S' = S'' \circ S$  für irgendeine Substitution  $S''$ .

# Kinds

Was mit unseren anderen Freunden, die Typkonstruktoren? Die sind ja eigentlich (syntaktische, termalgebraische) Funktionen. Die müssten doch eigentlich auch Typen haben, oder?

Hmmmmmm.

Zum Beispiel:

```
data MyType a = TC a
```

Wir haben natürlich  $\text{TC} :: a \rightarrow \text{MyType } a$ . Das ist wie eine normale Funktion. Aber was mit dem parametrisierten Typkonstruktor **MyType**? Müsste doch eigentlich eine Funktion von Typen zu Typen sein. Der nimmt ja ein Typargument  $\tau$  durch das formale Typparameter **a** und gibt für jeden Argumenttyp  $\tau$  den Typ **MyType**  $\tau$  zurück.

# Kinds

In der Tat: Es gibt einen “Typ der Typen”. Der heisst  $*$ . Ein “Typ von Typen” heißt ein *Kind*. (“Typ von Typen” riecht etwa nach Russell’s Paradoxon, nicht wahr). Ein *Kind* ist ein “Typ auf der nächsten Stufe”. Mit dieser Sprache der *Kinds* kommt man schon ganz weit:

$$\kappa ::= * \mid \kappa \rightarrow \kappa$$

Also, zum Beispiel:

```
MyType :: * -> *
```