Programmierung und Modellierung mit Haskell

Teil 8: Typsysteme & Typinferenz

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

4. Juni 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 8: Typsysteme & Typinferenz

1. Typsysteme
2. Typisierung
   - Grundlagen
   - Typregeln
   - Typherleitung
   - Ausblick
   - Strukturelle Typregeln
3. Polymorphie
   - Typsubstitutionen
   - Prinzipale Typisierung
   - Inferenzprobleme
4. Typinferenz
   - Unifikation
   - Hindley-Milner-Damas
5. Zusammenfassung Typsysteme

Typprüfung

Wir haben bisher behauptet:

Das **statische Typsystem** von Haskell prüft während dem Kompilieren alle Typen:

- Keine Typfehler und keine Typprüfung zur Laufzeit
- Kompiler findet bereits viele Programmierfehler
- Kompiler kann besser optimieren
- "Well-typed programs can't go wrong!" R. Milner 1934-2010

Heute klären wir:

- Was genau sind Typfehler?
- Was ist der Unterschied zwischen getypten und ungetypten Sprachen?
- Wie bestimmt man den Typ eines Programmes?

Typfehler

Offensichtlich machen folgende Ausdrücke keinen Sinn:

- `"Hello" * True`
- `0.23 + 'a'`
- `False && (\x -> "error")`
- `(*) 1 (+) 2 3 4`
- `(\f -> f 42) 69`
- `if 3 * x then 42 else "nothing"`

Ist dies wirklich *offensichtlich*?

Wie formalisieren wir die Intuition, welche Ausdrücke korrekt sind?

Typsysteme in der Praxis

In der Praxis gibt es durchaus unterschiedliche Ansätze zum Einsatz von Typsystemen in der Programmierung.

Was passiert bei der Auswertung von `(\f -> f 42) 69`?

Keine Typprüfung: `69` wird als Sprungaddresse interpretiert.
Systemabsturz oder schwere Betriebssystem-Ausnahme möglich.

Dynamische Typprüfung: Werte tragen zur Laufzeit Typ-Markierung *Ganzzahl*, *Funktion*, ...
Da `69` keine Funktion ist, wird bei der Ausführung eine Ausnahme im Laufzeitsystem geworfen.

Statische Typprüfung: Der Übersetzer weigert sich, ein Programm mit Typfehlern zu übersetzen. Es kommt nie zu derartigen Laufzeitfehlern.

Typsysteme in der Praxis

Unterschiedliche Ansätze zum Einsatz von Typsystemen:

Keine Typprüfung:

Assembler

- ⊕ Schnelle Ausführung; Viel Freiheit für Programmierer
- ⊖ Keinerlei Sicherheit; Schwere Fehler möglich

Dynamische Typprüfung:

LISP, Scheme, BASIC, *Skriptsprachen:* JavaScript, Python

- ⊕ Viel Freiheit für Programmierer
- ⊖ Ständige Typprüfung zur Laufzeit verlangsamt Ausführung

Statische Typprüfung:

Haskell, SML, Ocaml, Scala, *Eingeschränkt auch:* C, Java

- ⊕ Kompiler schliesst viele Fehler von vorn herein aus
- ⊖ Eingeschränkte Freiheit beim Programmieren:
  nicht alle korrekten Programme sind erlaubt;
  langwierige Fehlermeldungen nerven beim Kompilieren

λ-Kalkül

Zur Vereinfachung betrachten wir nur ein Fragment von Haskell:
Der λ-Kalkül (1936, Alonzo Church 1902–95) beschreibt den Kern funktionaler Sprachen.
Ein **Programmausdruck** e im Lambda-Kalkül ist:

e	$::=$	x	Variable	`foo`
	$\mid$	c	Konstante	`2.718`
	$\mid$	$e_1\ e_2$	Anwendung	`sqrt 5.0`
	$\mid$	$\lambda x.e$	Funktionsabstraktion	`\x -> e`

Der Lambda-Kalkül ist bereits eine **Turing-vollständig** Sprache.
Auswertung erfolgt lediglich durch Funktionsanwendung.

Zu Vereinfachung:

- Wir verwenden weiterhin die Haskell-Syntax
- Weitere Haskell-Ausdrücke nehmen wir nach Bedarf hinzu
- Infix-Operatoren schreiben wir hier meist in Präfix-Notation

`(+) 2 7` anstatt `2 + 7`

λ-Kalkül

λ-Kalkül: Variablen, Funktionsanwendung, Funktionsabstraktion

Beispiele Haskell's "`const 0 1`" entspricht:

$$((\backslash x \to (\backslash y \to x))\ 0)\ 1 \rightsquigarrow\ (\backslash y \to 0)\ 1 \rightsquigarrow\ 0$$

Haskell's "`($) id 1`" entspricht:

$$\begin{aligned} & (\backslash f \to (\backslash x\ \to f\ x))\ (\backslash y \to y)\ 1 \\ \rightsquigarrow\ & (\backslash x\ \to (\backslash y \to y)\ x)\ 1 \\ \rightsquigarrow\ & (\backslash y \to y)\ 1) \\ \rightsquigarrow\ & 1 \end{aligned}$$

Allgemein Auswertung wie gewohnt mit Substitutionsmodell:
β-Reduktion $(\backslash x \to e_1)\ e_2 \rightsquigarrow\ e_1[e_2/x]$

Termsubstitution: Wdh. 3.4

Wir definieren **Substitution für Terme** wir folgt, falls $x \neq y$:

$$
\begin{aligned}
x[e_0/x] &:= e_0 \\
y[e_0/x] &:= y \\
(e_1\ e_2)[e_0/x] &:= e_1[e_0/x]\ e_2[e_0/x] \\
(\backslash x\ \texttt{->}\ e_1)[e_0/x] &:= \backslash x\ \texttt{->}\ e_1 \\
(\backslash y\ \texttt{->}\ e_1)[e_0/x] &:= \backslash y\ \texttt{->}\ e_1[e_0/x] \text{ falls } y \notin \mathrm{FV}(e_0) \\
(\backslash y\ \texttt{->}\ e_1)[e_0/x] &:= \backslash y'\ \texttt{->}\ (e_1[y'/y])[e_0/x] \text{ mit } y' \text{ frische Variable}
\end{aligned}
$$

Bei Substitutionen muss man aufpassen, dass freie Variablen nicht versehentlich eingefangen werden; ggf. müssen gebundene Variablen umbenannt werden:
$(\backslash x\ \texttt{->}\ e)$ ist α-**äquivalent** zu $(\backslash y\ \texttt{->}\ e[y/x])$

Wir schreiben kurz $e_0[e_1/x, e_2/y]$ für $\big(e_0[e_1/x]\big)[e_2/y]$, d.h. mehrere Substitution arbeiten wir von links nach rechts ab.

Freie / Gebundene Variablen

Formale Definition: Freie Variablen

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} && \text{Variable} \\ \mathrm{FV}(c) &= \{\,\} && \text{Konstante} \\ \mathrm{FV}(e_1\ e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) && \text{Applikation} \\ \mathrm{FV}(\backslash x\ \texttt{->}\ e) &= \mathrm{FV}(e) \setminus \{x\} && \text{Abstraktion} \end{aligned}$$

Man sagt auch: “Das Lambda bindet die Variable”, d.h.
eine im Funktionsrumpf frei vorkommende Variable,
wird durch Funktionsabstraktion gebunden.

Eine Variable kann in einem Term gleichzeitig gebunden und frei vorkommen, z.B. $\mathrm{FV}\big((\backslash x\ \texttt{->}\ y\ x)\ (\backslash z\ \texttt{->}\ x)\big) = \{x, y\}$
Hier kommt x zuerst gebunden und dann ungebunden vor. Diese Vorkommen sind getrennt zu betrachten; es sind verschiedene Variablen, welche zufällig den gleichen Namen tragen.

Freie / Gebundene Variablen

Formale Definition: Freie Variablen

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} && \text{Variable} \\ \mathrm{FV}(c) &= \{\,\} && \text{Konstante} \\ \mathrm{FV}(e_1\ e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) && \text{Applikation} \\ \mathrm{FV}(\backslash x\ \texttt{->}\ e) &= \mathrm{FV}(e) \setminus \{x\} && \text{Abstraktion} \end{aligned}$$

Man sagt auch: “Das Lambda bindet die Variable”, d.h.
eine im Funktionsrumpf frei vorkommende Variable,
wird durch Funktionsabstraktion gebunden.

Eine Variable kann in einem Term gleichzeitig gebunden und frei vorkommen, z.B. $\mathrm{FV}\big((\backslash x\ \texttt{->}\ y\ x)\ (\backslash z\ \texttt{->}\ x)\big) = \{x, y\}$
Hier kommt x zuerst gebunden und dann ungebunden vor. Diese Vorkommen sind getrennt zu betrachten; es sind verschiedene Variablen, welche zufällig den gleichen Namen tragen.

Freie / Gebundene Variablen

Formale Definition: Freie Variablen

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} && \text{Variable} \\ \mathrm{FV}(c) &= \{\,\} && \text{Konstante} \\ \mathrm{FV}(e_1\ e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) && \text{Applikation} \\ \mathrm{FV}(\backslash x\ \texttt{->}\ e) &= \mathrm{FV}(e) \setminus \{x\} && \text{Abstraktion} \end{aligned}$$

Man sagt auch: “Das Lambda bindet die Variable”, d.h.
eine im Funktionsrumpf frei vorkommende Variable,
wird durch Funktionsabstraktion gebunden.

Eine Variable kann in einem Term gleichzeitig gebunden und frei vorkommen, z.B. $\mathrm{FV}\big((\backslash x\ \texttt{->}\ y\ x)\ (\backslash z\ \texttt{->}\ x)\big) = \{x, y\}$
Hier kommt x zuerst gebunden und dann ungebunden vor. Diese Vorkommen sind getrennt zu betrachten; es sind verschiedene Variablen, welche zufällig den gleichen Namen tragen.

Freie / Gebundene Variablen

Formale Definition: Freie Variablen

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} && \text{Variable} \\ \mathrm{FV}(c) &= \{\,\} && \text{Konstante} \\ \mathrm{FV}(e_1\ e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) && \text{Applikation} \\ \mathrm{FV}(\backslash x\ \texttt{->}\ e) &= \mathrm{FV}(e) \setminus \{x\} && \text{Abstraktion} \end{aligned}$$

Man sagt auch: “Das Lambda bindet die Variable”, d.h.
eine im Funktionsrumpf frei vorkommende Variable,
wird durch Funktionsabstraktion gebunden.

Eine Variable kann in einem Term gleichzeitig gebunden und frei vorkommen, z.B. $\mathrm{FV}\big((\backslash x\ \texttt{->}\ y\ x)\ (\backslash z\ \texttt{->}\ x)\big) = \{x, y\}$
Hier kommt x zuerst gebunden und dann ungebunden vor. Diese Vorkommen sind getrennt zu betrachten; es sind verschiedene Variablen, welche zufällig den gleichen Namen tragen.

Typausdrücke

Ein **Typausdruck** A is definiert durch:

				in Haskell
A, B	$::=$	$\alpha \mid \beta \mid \ldots$	Typvariable	`a`, `b`
	$\mid$	`Int` $\mid$ `Double` $\mid$ `String` $\mid \ldots$	Basistyp	
	$\mid$	$A \to B$	Funktionstyp	`Int -> Bool`

- Wir beschränken uns zur Vereinfachung auf Grund- und Funktionstypen; also keine Listen, keine Typklassen, etc.
- Funktionstypen sind weiterhin implizit rechts-geklammert:

$$A \to (B \to C) \;=\; A \to B \to C \;\neq\; (A \to B) \to C$$

- Eventuell verwenden wir Abkürzungen für Basistypen `I`, `D`, `S`, ... anstatt `Int`, `Double`, `String`, ...

Typzuweisung

Eine **Typzuweisung** $e::A$ ist ein Paar, bestehend aus einem Programmausdruck e und einen Typausdruck A, interpretiert als Aussage *"e hat Typ A"* In der Literatur $e:A$ statt $e::A$

`4`	::	`Int`	wahr
`true`	::	`Int`	falsch
`3.3`	::	`a`	falsch
`\x -> x - 1`	::	`Int-> Int`	wahr
`\x y -> x`	::	`a -> b -> a`	wahr
`\x y -> y`	::	`a -> b -> a`	falsch
`z (x + 0)`	::	`Bool`	???

- Geschlossene Typzuweisungs-Aussagen (d.h. *ohne freie Variablen*) können wahr oder falsch sein
- Der Typausdruck `z (x + 0)` enthält freie Variablen `x` und `z`, deren Typ wir nicht kennen

Typkontext

Ein **Typ(isierungs)kontext** Γ ist eine endliche Menge von Typzuweisungen (engl. *typing context* oder auch *environment*)

$$\Gamma = \{x_1 :: A_1, \ldots x_n :: A_n\} \qquad (n \geq 0)$$

wobei die Typvariablen x_i *paarweise verschieden* sein müssen.

$\{\texttt{x}::\texttt{Bool},\ \texttt{y}::\texttt{Int} \to \texttt{Int}\}$	*gültig*
$\{\texttt{z}::\texttt{Int} \to \texttt{Bool},\ \texttt{x}::\texttt{Int}\}$	*gültig*
$\{\texttt{h}::\texttt{Int} \to \texttt{Bool},\ \texttt{h}::\texttt{Int}\}$	*ungültig*

- Γ ist *endliche Abbildung* von Variablen x auf Typen $\Gamma(x)$
- Die Reihenfolge der Typzuweisungen in Γ ist unbedeutend
- Wir schreiben $\Gamma, x::A$ um $x::A$ in Γ *einzufügen*; falls $x \in \mathrm{dom}(\Gamma)$ dann ist $\Gamma, x::A$ undefiniert

Typurteil

Ein **Typurteil** (engl. *typing judgement*) ist eine Aussage der Form: *"Wenn die freien Variablen von e die in Γ angegebenen Typen haben, dann hat Ausdruck e den Typ A."* Wir schreiben kurz:

$$\Gamma \vdash e :: A$$

für die Aussage *"Ausdruck e hat Typ A in Kontext Γ"*

Formal ist $\_ \vdash \_ :: \_$ eine dreistellige Relation zwischen

1. Typkontext Γ
2. (Programm-)ausdruck e
3. Typ(-ausdruck) A

Γ, e, A sind **Metavariablen**
Der Programmausdruck e könnte für eine (Programm-)*variable* `foo` stehen.
Typ A könnte eine Typ*variable* α sein.

Beispiele:

$\{x :: \texttt{Int}\}$	$\vdash$	$\backslash y \to (+)\ x\ y$	$::$	$\texttt{Int} \to \texttt{Int}$	*gültig*
$\{x :: \texttt{Bool}\}$	$\vdash$	$\backslash y \to (+)\ x\ y$	$::$	$\texttt{Int} \to \texttt{Int}$	*ungültig*
$\{y :: \alpha\}$	$\vdash$	$\backslash x \to y$	$::$	$\beta \to \alpha$	*gültig*

Typregeln

Ein Typurteil ist nur dann gültig, wenn es durch **Typregeln** hergeleitet werden kann. Typregeln haben immer die Form:

$$\frac{P_1 \quad \cdots \quad P_n}{K} \text{ (Name der Typregel)}$$

Dabei sind $P_1 \ldots P_n$ die **Prämissen** der Typregel und K die **Konklusion**. Sind alle Prämissen gültig, dann auch die Konklusion.

- P_i und K sind Aussagen, hier meist Typurteile
- Typregeln ohne Prämissen heißen **Axiome**

  Z.B. für jede Konstante c gibt es ein Typaxiom.

Aus Platzgründen schreiben wir die Prämissen auch mal als "Wolke":

$$\frac{\begin{array}{ccc} P_1 & P_2 & P_3 \\ P_4 & P_5 & \end{array}}{K}$$

Typregel Var

Der Typ einer Variablen ist immer durch den Kontext gegeben:

$$\frac{(x::A) \in \Gamma}{\Gamma \vdash x::A} \quad (\textsc{Var})$$

Äquivalent wird die Regel in der Literatur auch oft so verfasst:

$$\frac{}{\Gamma, x::A \vdash x::A} \quad (\textsc{Var})$$

wobei die Kurzschreibweise $\Gamma, x::A$ ja gerade für einen Typkontext steht, welcher u.a. Variable x auf Typ A abbildet.

Beispiele:

$\{x::\texttt{Double}\}$	$\vdash$	$x::\texttt{Double}$	*(gültig)*
$\{x::\texttt{Double}, y::\texttt{Int}, z::\texttt{Bool}\}$	$\vdash$	$y::\texttt{Int}$	*(gültig)*
$\{x::\texttt{Double}, y::\texttt{Int}, z::\texttt{Bool}\}$	$\vdash$	$y::\texttt{Double}$	$\lightning$ *falscher Typ*
$\{x::\texttt{Int}\}$	$\vdash$	$y::\texttt{Int}$	$\lightning$ *ungebundene Variable*

Typregel Const

Für jede Konstante eines jeden Basistypen definieren wir ein **Axiom** (Regel ohne Prämisse), welches diese Konstante erkennt:

$$\frac{}{\Gamma \vdash \texttt{True}::\texttt{Bool}}\ (\textsc{True}) \qquad \frac{}{\Gamma \vdash \texttt{False}::\texttt{Bool}}\ (\textsc{False})$$

Dabei ist der Kontext Γ hier beliebig, d.h. wir definieren ein **Axiomenschema** für jedes Γ.

Mehrere Konstanten fassen wir mit folgenden Schemen zusammen:

$$\frac{c \text{ ist eine Integer Konstante}}{\Gamma \vdash c::\texttt{Int}}\ (\textsc{Int}) \qquad \frac{c \text{ ist eine Double Konstante}}{\Gamma \vdash c::\texttt{Double}}\ (\textsc{Double})$$

Es ist sowohl $\{\} \vdash 5::\texttt{Int}$ als auch $\{\} \vdash 5::\texttt{Double}$ herleitbar.
So lange wir Typklassen ignorieren ist dies unproblematisch.

Typregel Funktionsanwendung

Anwendung einer Funktion e_1 mit Typ $A \to B$ auf Argument e_2 von Typ A liefert ein Ergebnis des Typs B:

$$\frac{\Gamma \vdash e_1 :: A \to B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B} \quad (\textsc{App})$$

Diese Regel ist nicht anwendbar, falls:

- e_1 nicht von einem Funktionstypen ist — e_1 hat keinen Pfeil-Typ
- e_2 nicht nicht zum Definitionsbereich passt — e_2 hat nicht Typ A

Beispiel:

$$\frac{\{x :: \texttt{String} \to \texttt{Bool}, y :: \texttt{String}\} \vdash x :: \texttt{String} \to \texttt{Bool} \qquad \{x :: \texttt{String} \to \texttt{Bool}, y :: \texttt{String}\} \vdash y :: \texttt{String}}{\{x :: \texttt{String} \to \texttt{Bool}, y :: \texttt{String}\} \vdash x\ y :: \texttt{Bool}} \quad (\textsc{App})$$

Typregel Funktionsabstraktion

Die Typregel für anonyme Funktionsabstraktion lautet:

$$\frac{\Gamma, x::A \vdash e::B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

"Hat der Programmausdruck e den Typ B unter der Annahme, dass x den Typ A hat; so hat der Programmausdruck $\backslash x \to e$ den Typ $A \to B$ ohne diese Annahme, dass x den Typ A hat."

Man nennt e den **Funktionsrumpf** der Funktion $\backslash x \to e$,
im Funktionsrumpf kommt die abstrahierte Variable x *frei* vor,
in dem Programmausdruck $\backslash x \to e$ *gebunden*.
Ein Lambda **bindet** also die darauf folgende Variable.

Instantiierung einer Typregel

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Bei der Anwendung dieser Typregel instantiieren wie die **Metavariablen**: Γ, x, e, A, B. Alles andere beleibt unverändert!

- Metavariable x wird ersetzt durch eine Programmvariable
- Γ wird ersetzt durch einen beliebigen Typkontext;
  die für x gewählte Variable darf nicht in Γ enthalten sein!
- Für e wählen wir einen Programmausdruck
- Für A und B wählen wir beliebige Typen
  Bei Polymorphie verwenden wir zur Unterscheidung zu Metavariablen griechische Buchstaben $\alpha, \beta, \ldots$
  entsprechen Kleinbuchstaben in Typen von Haskell

Beispiel:

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Instantiierung einer Typregel

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Bei der Anwendung dieser Typregel instantiieren wie die **Metavariablen**: Γ, x, e, A, B. Alles andere beleibt unverändert!

- Metavariable x wird ersetzt durch eine Programmvariable
- Γ wird ersetzt durch einen beliebigen Typkontext;
  die für x gewählte Variable darf nicht in Γ enthalten sein!
- Für e wählen wir einen Programmausdruck
- Für A und B wählen wir beliebige Typen
  Bei Polymorphie verwenden wir zur Unterscheidung zu Metavariablen griechische Buchstaben $\alpha, \beta, \ldots$
  entsprechen Kleinbuchstaben in Typen von Haskell

Beispiel:

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Instantiierung einer Typregel

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Bei der Anwendung dieser Typregel instantiieren wie die **Metavariablen**: Γ, x, e, A, B. Alles andere beleibt unverändert!

- Metavariable x wird ersetzt durch eine Programmvariable
- Γ wird ersetzt durch einen beliebigen Typkontext;
  die für x gewählte Variable darf nicht in Γ enthalten sein!
- Für e wählen wir einen Programmausdruck
- Für A und B wählen wir beliebige Typen
  Bei Polymorphie verwenden wir zur Unterscheidung zu Metavariablen griechische Buchstaben $\alpha, \beta, \ldots$
  entsprechen Kleinbuchstaben in Typen von Haskell

Beispiel:

$$\frac{\Gamma, wuz::A \vdash e :: B}{\Gamma \vdash \backslash wuz \to e :: A \to B} \quad (\textsc{Abs})$$

Instantiierung einer Typregel

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Bei der Anwendung dieser Typregel instantiieren wie die **Metavariablen**: Γ, x, e, A, B. Alles andere beleibt unverändert!

- Metavariable x wird ersetzt durch eine Programmvariable `wuz`
- Γ wird ersetzt durch einen beliebigen Typkontext;
  die für x gewählte Variable darf nicht in Γ enthalten sein!
- Für e wählen wir einen Programmausdruck
- Für A und B wählen wir beliebige Typen
  Bei Polymorphie verwenden wir zur Unterscheidung zu Metavariablen griechische Buchstaben $\alpha, \beta, \ldots$
  entsprechen Kleinbuchstaben in Typen von Haskell

Beispiel:

$$\frac{\{wuz::A\} \vdash e :: B}{\{\,\} \vdash \backslash wuz \to e :: A \to B} \quad (\textsc{Abs})$$

Instantiierung einer Typregel

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Bei der Anwendung dieser Typregel instantiieren wie die **Metavariablen**: Γ, x, e, A, B. Alles andere beleibt unverändert!

- Metavariable x wird ersetzt durch eine Programmvariable `wuz`
- Γ wird ersetzt durch einen beliebigen Typkontext; $\{\}$
  die für x gewählte Variable darf nicht in Γ enthalten sein!
- Für e wählen wir einen Programmausdruck
- Für A und B wählen wir beliebige Typen
  Bei Polymorphie verwenden wir zur Unterscheidung zu Metavariablen griechische Buchstaben $\alpha, \beta, \ldots$
  entsprechen Kleinbuchstaben in Typen von Haskell

Beispiel:

$$\frac{\{wuz::A\} \vdash wuz + 1 :: B}{\{\} \vdash \backslash wuz \to wuz + 1 :: A \to B} \quad (\textsc{Abs})$$

Instantiierung einer Typregel

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Bei der Anwendung dieser Typregel instantiieren wie die **Metavariablen**: Γ, x, e, A, B. Alles andere beleibt unverändert!

- Metavariable x wird ersetzt durch eine Programmvariable `wuz`
- Γ wird ersetzt durch einen beliebigen Typkontext; $\{\}$
  die für x gewählte Variable darf nicht in Γ enthalten sein!
- Für e wählen wir einen Programmausdruck `wuz + 1`
- Für A und B wählen wir beliebige Typen
  Bei Polymorphie verwenden wir zur Unterscheidung zu Metavariablen griechische Buchstaben $\alpha, \beta, \ldots$
  entsprechen Kleinbuchstaben in Typen von Haskell

Beispiel:

$$\frac{\{wuz::\texttt{Int}\} \vdash wuz + 1 :: B}{\{\} \vdash \backslash wuz \to wuz + 1 :: \texttt{Int} \to B} \quad (\textsc{Abs})$$

Instantiierung einer Typregel

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

Bei der Anwendung dieser Typregel instantiieren wie die **Metavariablen**: Γ, x, e, A, B. Alles andere beleibt unverändert!

- Metavariable x wird ersetzt durch eine Programmvariable `wuz`
- Γ wird ersetzt durch einen beliebigen Typkontext; $\{\}$
  die für x gewählte Variable darf nicht in Γ enthalten sein!
- Für e wählen wir einen Programmausdruck `wuz + 1`
- Für A und B wählen wir beliebige Typen `Int`
  Bei Polymorphie verwenden wir zur Unterscheidung zu Metavariablen griechische Buchstaben $\alpha, \beta, \ldots$
  entsprechen Kleinbuchstaben in Typen von Haskell

Beispiel:

$$\frac{\{wuz::\texttt{Int}\} \vdash wuz + 1 :: \texttt{Int}}{\{\} \vdash \backslash wuz \to wuz + 1 :: \texttt{Int} \to \texttt{Int}} \quad (\textsc{Abs})$$

Typherleitung

Eine **Typherleitung** / **Typbeweis** ist ein endlicher Baum, wobei

- alle Blätter Typaxiome sind;
- alle Knoten derart Instanzen von Typregeln sind, dass die Beschriftung des Knotens die Konklusion ist, und die Kinder des Knotens die Prämissen sind.

Die Beschriftung des Wurzelknotens ist die **hergeleitete** Aussage.

Eine Typaussage ist **herleitbar**, wenn sie eine Herleitung hat.

Eine Typherleitung ist letztendlich nur eine verkette Anwendung von Typregeln.

Beispiel Herleitungsbaum

Eine Herleitung als Baum dargestellt:

$$\dfrac{\dfrac{\dfrac{}{\{x{::}\alpha,\ f{::}\alpha \to \beta\} \vdash f :: \alpha \to \beta}\ \textsc{Var} \quad \dfrac{}{\{x{::}\alpha,\ f{::}\alpha \to \beta\} \vdash x :: \alpha}\ \textsc{Var}}{\{x{::}\alpha,\ f{::}\alpha \to \beta\} \vdash f\ x :: \beta}\ \textsc{App}}{\{x{::}\alpha\} \vdash \backslash f \to f\ x :: (\alpha \to \beta) \to \beta}\ \textsc{Abs}}{\{\} \vdash \backslash x \to (\backslash f \to f\ x) :: \alpha \to (\alpha \to \beta) \to \beta}\ \textsc{Abs}$$

Den Namen der angewendeten Regel schreiben wir an den rechten Rand, damit man besser nachvollziehen kann, was gemacht wurde.

Hinweis: In Haskell könnten wir anstatt $\backslash x \to (\backslash f \to f\ x)$ auch einfach $\backslash x\ f \to f\ x$ schreiben, doch unsere Typregel $(\textsc{Abs})$ haben wir zur Vereinfachung nur für ein Argument definiert.

Beispiel Herleitungen

Die gleiche Herleitung noch einmal in linearer Schreibweise:

Nr	Konklusion	Regel(Prämissen)
#1	$\{\} \vdash \backslash x \to (\backslash f \to f\ x) :: \alpha \to (\alpha \to \beta) \to \beta$	Abs(#2)
#2	$\{x :: \alpha\} \vdash \backslash f \to f\ x :: (\alpha \to \beta) \to \beta$	Abs(#3)
#3	$\{x :: \alpha,\ f :: \alpha \to \beta\} \vdash f\ x :: \beta$	App(#4, #5)
#4	$\{x :: \alpha,\ f :: \alpha \to \beta\} \vdash x :: \alpha$	Var
#5	$\{x :: \alpha,\ f :: \alpha \to \beta\} \vdash f :: \alpha \to \beta$	Var

- Jede Konklusion steht alleine in einer nummerierte Zeile.
- Jede Konklusion wird begründet durch Angabe der Typregel und die Zeilennummern der verwendeten Prämissen.
- Die Reihenfolge der Zeilen ist prinzipiell egal; in der ersten Zeile schreiben wir die zu beweisende Aussage hin.
  Hier: $\{\} \vdash \backslash x \to (\backslash f \to f\ x) :: \alpha \to (\alpha \to \beta)$

Typherleitung Beispiel

An der Tafel konstruieren wir eine Typherleitung für das Typurteil:

$$\{(+)::\texttt{Int} \to \texttt{Int} \to \texttt{Int}, z::\alpha\} \vdash (+)\ 4\ ((\backslash x\ \texttt{->}\ 1)\ z) :: \texttt{Int}$$

Bisher behandelte Typregeln

Folgende Typregeln wurden bisher in der Vorlesung behandelt:

$$\frac{}{\Gamma, x::A \vdash x::A} \quad \text{(Var)}$$

$$\frac{c \text{ ist eine Integer Konstante}}{\Gamma \vdash c :: \texttt{Int}} \quad \text{(Int)}$$

$$\frac{\Gamma \vdash e_1 :: A \to B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B} \quad \text{(App)}$$

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad \text{(Abs)}$$

Typregel Konditional

Wir können weitere Ausdrücke mit neuen Typregeln behandeln:

Neuer Programmausdruck: `if` e_1 `then` e_2 `else` e_3

Neue Typregel:

$$\frac{}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A}\ (\textsc{Cond})$$

Bedeutung:

- Bedingung e_1 muss Typ `Bool` haben
- Zweige e_2 und e_3 müssen vom gleichen Typ wie Gesamtausdruck sein

Typregel Konditional

Wir können weitere Ausdrücke mit neuen Typregeln behandeln:

Neuer Programmausdruck: `if` e_1 `then` e_2 `else` e_3

Neue Typregel:

$$\frac{\Gamma \vdash e_1 :: \texttt{Bool}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A} \quad (\textsc{Cond})$$

Bedeutung:

- Bedingung e_1 muss Typ `Bool` haben
- Zweige e_2 und e_3 müssen vom gleichen Typ wie Gesamtausdruck sein

Typregel Konditional

Wir können weitere Ausdrücke mit neuen Typregeln behandeln:

Neuer Programmausdruck: `if` e_1 `then` e_2 `else` e_3

Neue Typregel:

$$\frac{\Gamma \vdash e_1 :: \texttt{Bool} \qquad \Gamma \vdash e_2 :: ? \qquad \Gamma \vdash e_3 :: ?}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A}\ (\textsc{Cond})$$

Bedeutung:

- Bedingung e_1 muss Typ `Bool` haben
- Zweige e_2 und e_3 müssen vom gleichen Typ wie Gesamtausdruck sein

Typregel Konditional

Wir können weitere Ausdrücke mit neuen Typregeln behandeln:

Neuer Programmausdruck: `if` e_1 `then` e_2 `else` e_3

Neue Typregel:

$$\frac{\Gamma \vdash e_1 :: \texttt{Bool} \qquad \Gamma \vdash e_2 :: A \qquad \Gamma \vdash e_3 :: A}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A}\ (\textsc{Cond})$$

Bedeutung:

- Bedingung e_1 muss Typ `Bool` haben
- Zweige e_2 und e_3 müssen vom gleichen Typ wie Gesamtausdruck sein

Safety = Progress + Preservation

Eine Typherleitung ist ein formaler Beweis, dass ein Typausdruck einen bestimmten Typ hat.

Was nutzt das?

Wir behaupteten: "Well-typed programs can't go wrong"
Dies beweist man üblicherweise in 2 Schritten:

Progress

Jeder wohl-typisierte Programmausdruck ist entweder ein Wert oder er kann weiter ausgewertet werden.

Preservation

Auswertung verändert den Typ eines Programmausdrucks nicht.

Für diese Beweise müssen wir das Substitutionsmodell durch Auswerteregeln (**Operationale Semantik**) formalisieren, wie z.B.

$$\frac{e_1[e_2/x] \leadsto v}{(\backslash x \ \texttt{->} \ e_1)\ e_2 \leadsto v} \qquad (\leadsto\text{-App})$$

Beweisidee:

Solche Beweise werden üblicherweise mit Induktion über die Länge der Typherleitung geführt:
Um zu zeigen, dass `if` e_1 `then` e_2 `else` e_3 ausgewertet werden kann, darf man annehmen dann, dass e_1 zu einem Wert ausgewertet werden kann, da die Typherleitung für $\Gamma \vdash e_1 : \texttt{Bool}$ ja ein Schritt kleiner sein muss; usw.

$$\frac{e_1 \rightsquigarrow \texttt{True} \qquad e_2 \rightsquigarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow v}\ (\rightsquigarrow\text{-If-T}) \qquad \frac{e_1 \rightsquigarrow \texttt{False} \qquad e_3 \rightsquigarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow v}\ (\rightsquigarrow\text{-If-F})$$

$$\frac{\Gamma \vdash e_1 :: \texttt{Bool} \qquad \Gamma \vdash e_2 :: A \qquad \Gamma \vdash e_3 :: A}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A}\ (\text{Cond})$$

- Auswerteregeln und Typregeln müssen korrespondieren.
- Auswerteregeln und Implementation müssen übereinstimmen.

Bisher behandelte Typregeln

Folgende Typregeln wurden bisher in der Vorlesung behandelt:

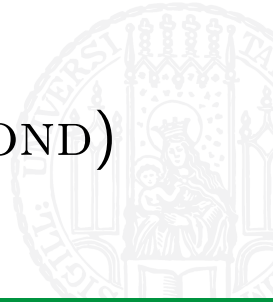
$$\frac{}{\Gamma, x::A \vdash x::A} \quad \text{(Var)}$$

$$\frac{c \text{ ist eine Integer Konstante}}{\Gamma \vdash c :: \texttt{Int}} \quad \text{(Int)}$$

$$\frac{\Gamma \vdash e_1 :: A \to B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B} \quad \text{(App)}$$

$$\frac{\Gamma, x::A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad \text{(Abs)}$$

$$\frac{\Gamma \vdash e_1 :: \texttt{Bool} \qquad \Gamma \vdash e_2 :: A \qquad \Gamma \vdash e_3 :: A}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A} \quad \text{(Cond)}$$



Curry-Howard-Isomorphismus

Typherleitungen sind sehr ähnlich zur mathematischen Beweisführung im Hilbert-Stil, welche ebenfalls Herleitungsbäume verwendet.

Curry-Howard-Isomorphismus

Ein Typ kann als logische Formel aufgefasst werden.
Gibt es einen Programmausdruck zu einem Typ, so ist die entsprechende logische Formel *intuitionistisch* beweisbar.

- Typ $A \to B$ entspricht der logischen Implikation,
- Typ (A, B) entspricht dem logischen "und", usw.

Beobachtet durch Curry & Feys (1958) und Howard (1969)

Anwendung Übertragung von Erkenntnissen zwischen Fachgebieten, Konstruktion von Beweisassistenten, Automatische Programmextraktion aus intuitionistischen Beweisen

Curry-Howard-Isomorphismus

Typherleitungen sind sehr ähnlich zur mathematischen Beweisführung im Hilbert-Stil, welche ebenfalls Herleitungsbäume verwendet.

Intuitionistische oder auch **Konstruktive Logik**: Axiom vom ausgeschlossenen Dritten "$\vdash A \vee \neg A$" gilt nicht.

Curry-Howard-Isomorphismus

Ein Typ kann als logische Formel aufgefasst werden.
Gibt es einen Programmausdruck zu einem Typ, so ist die entsprechende logische Formel *intuitionistisch* beweisbar.

- Typ $A \to B$ entspricht der logischen Implikation,
- Typ (A, B) entspricht dem logischen "und", usw.

Beobachtet durch Curry & Feys (1958) und Howard (1969)

Anwendung Übertragung von Erkenntnissen zwischen Fachgebieten, Konstruktion von Beweisassistenten, Automatische Programmextraktion aus intuitionistischen Beweisen

Kontexterweiterung

Eine Typaussage bleibt gültig, wenn wir den Kontext **erweitern**.

Beispiel:

$$\begin{array}{lll} \{x::\alpha\} & \vdash & \backslash f \to f\ x :: (\alpha \to \beta) \to \beta \\ \{x::\alpha, y::\gamma\} & \vdash & \backslash f \to f\ x :: (\alpha \to \beta) \to \beta \\ \{f::\texttt{Int} \to \texttt{Bool}, x::\alpha, y::\gamma\} & \vdash & \backslash f \to f\ x :: (\alpha \to \beta) \to \beta \end{array}$$

Lemma (Kontexterweiterung/Abschwächung)

Wenn $\Gamma \subseteq \Gamma'$ und $\Gamma \vdash e::A$ herleitbar ist, dann auch $\Gamma' \vdash e::A$.

Beweisbar durch Induktion über die Länge der Herleitung.

Daraus leiten wir folgende **strukturelle Typregel** ab:

$$\frac{\Gamma' \supseteq \Gamma \qquad \Gamma \vdash e::A}{\Gamma' \vdash e::A} \quad \text{(Weak)}$$

Die Regel ist *jederzeit anwendbar*, da sie unabhängig von e ist.

Weakening

Kontexterweiterungen sind harmlos:

$$\frac{\Gamma' \supseteq \Gamma \qquad \Gamma \vdash e::A}{\Gamma' \vdash e::A} \quad (\textsc{Weak})$$

Intuitiv: *"Es schadet nie, mehr zu wissen als nötig"*

Die Typregel Weak ist **zulässig** zu den bisher vorgestellten Typregeln, d.h. wir können mit dieser Typregel auch nicht mehr Typurteile beweisen, als ohne diese Typregel. D.h. die Regel kann Herleitungen lediglich vereinfachen.

Zulässig vs. Herleitbar

In Systemen des **natürlichen Schließens** unterscheidet man bei "harmlosen" Regelerweiterungen üblicherweise folgende Varianten:

Zulässig Eine Regel ist **zulässig** (engl. *admissible*), falls deren Hinzunahme keine neuen Herleitungen ermöglicht.

Herleitbar Eine Regel ist **herleitbar** (engl. *derivable*), wenn sie lediglich aus der aneinandergereihten Anwendung von verschieden existierenden Regeln zusammengesetzt ist.

Merke: Eine herleitbare Regel ist immer auch zulässig; aber umgekehrt gilt dies nicht immer!

Solche Regelerweiterungen dienen also nur zur Vereinfachung der Beweisführung, aber erlauben nicht den Beweis neuer Aussagen!

Exchange

Da wir Typkontexte als Mengen von Typzuweisungen definiert haben, ist die Reihenfolge der Typannahmen bedeutungslos.

Dies können wir ebenfalls durch eine zulässige Regel ausdrücken:

$$\frac{\Gamma, x::A, y::B, \Delta \vdash e::C}{\Gamma, y::B, x::A, \Delta \vdash e::C} \quad \text{(Exchange)}$$

Kontraktion

Mehrfachverwendung von Variablen ist harmlos:

$$\frac{\Gamma, x::A, y::A \vdash e::C \qquad z \notin \mathrm{FV}(e)}{\Gamma, z::A \vdash e[z/x, z/y]::C}\ (\textsc{Contraction})$$

Zur Erinnerung:

- $z \notin \mathrm{FV}(e)$ bedeutet, dass Variable z in Term e nicht frei vorkommt; also ist z eine neue/frische Variable.

- $e[z/x, z/y]$ bezeichnet den Term e, bei den alle freie Vorkommen von x und y ersetzt werden (siehe Folie 8.9).
  Merke: z/x bedeutet: z nimmt einen Stock / und erschlägt x

Strukturelle Typregeln

- Zusätzliche Annahmen sind harmlos:

$$\frac{\Gamma' \supseteq \Gamma \qquad \Gamma \vdash e::A}{\Gamma' \vdash e::A} \quad \text{(Weak)}$$

- Reihenfolge der Annahmen ist bedeutungslos:

$$\frac{\Gamma, x::A, y::B, \Delta \vdash e::C}{\Gamma, y::B, x::A, \Delta \vdash e::C} \quad \text{(Exchange)}$$

- Eine Variable kann mehrfach verwendet werden:

$$\frac{\Gamma, x::A, y::A \vdash e::C \qquad z \notin \mathrm{FV}(e)}{\Gamma, z::A \vdash e[z/x, z/y]::C} \quad \text{(Contraction)}$$

Üblicherweise gelten diese drei strukturellen Typregeln *implizit*, wenn nichts anderes angegeben wurde. Man kann aber auch Typsysteme bauen, in denen diese Typregeln nicht gelten.

Strukturelle Typregeln

- Zusätzliche Annahmen sind harmlos:

$$\frac{\Gamma' \supseteq \Gamma \qquad \Gamma \vdash e::A}{\Gamma' \vdash e::A} \text{ (Weak)}$$

Beispiel:
Wenn man mit einem Typsystem zeigen will, dass ein Programm alle geöffneten Dateien vor Ende ordentlich schließt, könnte man Weak und Contraction für den Typ der File-Handles verbieten und nur für harmlose Typen wir `Int` erlauben; dann spricht man von einem *linearen* Typsystem.

$$\frac{\Gamma, x::A, y::A \vdash e::C \qquad z \notin \mathrm{FV}(e)}{\Gamma, z::A \vdash e[z/x, z/y]::C} \text{ (Contraction)}$$

Üblicherweise gelten diese drei strukturellen Typregeln *implizit*, wenn nichts anderes angegeben wurde. Man kann aber auch Typsysteme bauen, in denen diese Typregeln nicht gelten.

Polymorphie

Wie wir wissen, können Ausdrücke mehr als einen Typ haben, z.B. für den Programmausdruck `f x` sind herleitbar:

$$\begin{array}{lll}
\{x{::}\texttt{Int},\ f{::}\texttt{Int} \to \texttt{Int}\} & \vdash & f\,x\ ::\ \texttt{Int} \\
\{x{::}\texttt{Bool},\ f{::}\texttt{Bool} \to \texttt{Int}\} & \vdash & f\,x\ ::\ \texttt{Int} \\
\{x{::}\alpha,\ f{::}\alpha \to \alpha\} & \vdash & f\,x\ ::\ \alpha \\
\{x{::}\alpha,\ f{::}\alpha \to \beta\} & \vdash & f\,x\ ::\ \beta
\end{array}$$

Letzte hier ist **allgemeinste Typisierung** (engl. *principal typing*): jede andere Typisierung von `f x` erhält man daraus durch Einsetzen von Typen für die Typvariablen α und β (**Instanziierung**).

Zum Beispiel für 2.Zeile:

$$\{x{::}\alpha,\ f{::}\alpha \to \beta\}[\texttt{Bool}/\alpha,\ \texttt{Int}/\beta] \vdash f\,x\ ::\ \beta[\texttt{Bool}/\alpha,\ \texttt{Int}/\beta]$$

Man kann beweisen: Allgemeinste Typisierung eines Ausdrucks ist bis auf Umbenennung von Typvariablen eindeutig.

Typsubstitutionen

Eine **Typsubstitution** σ ist eine endliche Abbildung von Typvariablen α auf Typausdrücke A. *Beispiele:*

$$\begin{aligned}
\sigma_1 &= [\texttt{Bool}/\alpha, \texttt{Int}/\beta] \\
&= [\texttt{Bool}/\alpha, \texttt{Int}/\beta, \gamma/\gamma] \\
&= [\texttt{Bool}/\alpha, \texttt{Int}/\beta, \gamma/\gamma, \delta/\delta] \ldots \\
\sigma_2 &= [\beta/\alpha, (\beta \to \texttt{Bool})/\beta, \texttt{Int}/\gamma] \\
\sigma_3 &= [\gamma/\alpha, \delta/\beta] \\
\sigma_4 &= [\beta/\alpha, \alpha/\beta]
\end{aligned}$$

Für die Anwendung $A\sigma$ einer Substitution σ auf Typen A gilt z.B.:

$$\begin{aligned}
\texttt{Int}\,\sigma &= \texttt{Int} \\
(A \to B)\sigma &= A\sigma \to B\sigma \\
\alpha[\texttt{Bool}/\alpha, \texttt{Int}/\beta] &= \texttt{Bool}
\end{aligned}$$

Da bei uns alle Typvariablen frei sind, gibt es keine vergleichbaren Probleme wir bei der Termsubstitution.

Instanziierung

Wir definieren die Anwendung einer Substitution σ auf einen Typkontext Γ punktweise:

$$(\Gamma\sigma)(x) := (\Gamma(x))\sigma$$

Beispiel

$$\{x{::}\texttt{Int},\ y{::}\alpha \to \beta,\ z{::}\alpha\}[\gamma/\alpha, \texttt{Int}/\beta, \texttt{Bool}/\gamma] \quad = \\ \{x{::}\texttt{Int},\ y{::}\texttt{Bool} \to \texttt{Int},\ z{::}\texttt{Bool}\}$$

Es ist beweisbar, dass Typurteile unter Substitution gültig bleiben:

Lemma (Typerhaltung unter Substitution)

Wenn $\Gamma \vdash e :: A$, *dann* $\Gamma\sigma \vdash e :: A\sigma$.

Wir leiten daraus erneut eine zulässige Typregel ab:

$$\frac{\Gamma \vdash e :: A}{\Gamma\sigma \vdash e :: A\sigma} \quad \text{(Subst)}$$

Substitutionskomposition

Die **Komposition** $\sigma\sigma'$ zweier Substitutionen definieren wir wieder von links nach rechts:

$$A(\sigma_1\sigma_2) := (A\sigma_1)\sigma_2$$

Eine Komposition können wir oft vereinfachen:

$$\begin{array}{rcl}
\alpha[(\alpha \to \beta)/\alpha][\mathtt{Int}/\alpha,\ \mathtt{Bool}/\beta] & = & (\alpha \to \beta)[\mathtt{Int}/\alpha,\ \mathtt{Bool}/\beta] \\
 & = & \mathtt{Int} \to \mathtt{Bool} \\
\beta[(\alpha \to \beta)/\alpha][\mathtt{Int}/\alpha,\ \mathtt{Bool}/\beta] & = & \beta[\mathtt{Int}/\alpha,\ \mathtt{Bool}/\beta] \\
 & = & \mathtt{Bool} \\
\hline
[(\alpha \to \beta)/\alpha][\mathtt{Int}/\alpha,\ \mathtt{Bool}/\beta] & = & [(\mathtt{Int} \to \mathtt{Bool})/\alpha,\ \mathtt{Bool}/\beta]
\end{array}$$

- Komposition ist **assoziativ**, $\sigma_1(\sigma_2\sigma_3) = (\sigma_1\sigma_2)\sigma_3$, aber nicht kommutativ $\sigma_1\sigma_2 \neq \sigma_2\sigma_1$
- Als **neutrales Element** der Komposition haben wir die leere **Identität**ssubstitution $\mathrm{id} = [\,]$. Es gilt $\mathrm{id}\,\sigma = \sigma$ und $\sigma\,\mathrm{id} = \sigma$.

Prinzipale Typisierung

Typisierung $\Gamma \vdash e :: A$ eines Ausdrucks e ist die **allgemeinste** oder auch **prinzipale** Typisierung, falls es für jede Typisierung $\Gamma' \vdash e :: A'$ von e eine Substitution σ gibt, so dass $A' = A\sigma$ und $\Gamma' \supseteq \Gamma\sigma$.

$$
\begin{array}{llll}
\{x{::}\alpha,\ f{::}\alpha \to \beta\} & \vdash\ f\,x\ ::\ \beta & & \text{prinzipal} \\
\{x{::}\texttt{Int},\ f{::}\texttt{Int} \to \texttt{Int}\} & \vdash\ f\,x\ ::\ \texttt{Int} & & \sigma = [\texttt{Int}/\alpha, \texttt{Int}/\beta] \\
\{x{::}\alpha,\ f{::}\alpha \to \beta,\ y{::}\gamma\} & \vdash\ f\,x\ ::\ \beta & & \sigma = \mathrm{id} \\
\{x{::}\gamma,\ f{::}\gamma \to \beta\} & \vdash\ f\,x\ ::\ \beta & & \sigma = [\gamma/\alpha] \quad \text{(auch prinzipal)}
\end{array}
$$

- Mann kann beweisen, dass jeder Programmausdruck unserer eingeschränkten Sprache eine prinzipale Typisierung hat!

  Für volles Haskell gilt dies aber nicht mehr.
- Prinzipaler Typ ist eindeutig, falls er existiert.
- Die Berechnung einer prinzipalen Typisierung für einen Programmausdruck ist oft **unentscheidbar**, d.h. es ist kein vollständiges Verfahren mit endlicher Laufzeit bekannt.

Monomorphismus Einschränkung

Bei GHC kann die Suche nach allgemeinen Typen eingeschränkt werden:

```
> :set -XMonomorphismRestriction
> :t show
show :: Show a => a -> String
> let f1 = show
f1 :: () -> String
> let f2 x = show x
f2 :: Show a => a -> String
> let f3 = \x -> show x
f3 :: () -> String
```

. . . oder erweitert werden (Standardeinstellung kann variieren!)

```
> :set -XNoMonomorphismRestriction
> let f4 = show
f4 :: Show a => a -> String
```

Monomorphismus Einschränkung

Diese Option hat aber auch Nachteile:

```
lenTwo = (len,len)
  where
    len = Data.List.genericLength xs
```

Für den Typ `lenTwo :: Num t => [b] -> (t, t)` muss die Länge nur einmal berechnet werden, aber
für den Typ `lenTwo :: (Num a, Num b) => [c] -> (a,b)` muss die Länge zwei Mal berechnet werden, da die aus der Typklasse abgeleitete Operation (hier die Addition) in jeder Typklasse ja anders implementiert sein kann.

Wenn man Typsignaturen explizit angibt, dann braucht man diese Option ohnehin nicht, da Haskell dann einfach nur den gegebenen Typ überprüft.

Polymorphe Rekursion

Haskell erlaubt **polymorphe Rekursion**, d.h. rekursive Aufrufe mit einer anderen Instantiierung der polymorphen Typparameter.

Polymorphe Rekursion ist *problematisch für Typinferenz*. Es wurde bewiesen, dass Typinferenz für polymorphe Rekursion unentscheidbar ist.

Entscheidbar: Algorithmus kann in endlicher Zeit korrekt antworten.

Beispiel

```
data PowerList a = Zero a | Succ (PowerList (a,a))

pl1 = Succ $ Succ $ Succ $ Zero (((1,2),(3,4)),((5,6),(7,8)))

plLength :: PowerList a -> Int -- GHC kann Typ nicht inferieren!
plLength (Zero _ ) = 1
plLength (Succ pl) = 2 * plLength pl
```

Im Rumpf der Definition für `plLength :: PowerList a -> Int` findet ein rekursiver Aufruf mit Typ `PowerList (a,a) -> Int` statt.

Spracherweiterungen

Statische Typsysteme sind vermutlich die am weitesten verbreitete Form der **automatischen Programmverifikation**.
Die meisten typisierbaren Programme sind sinnvoll und umgekehrt!

Problem:

- Nicht alle sinnvolle Programme sind typisierbar
- Nicht alle typisierbaren Programme sind sinnvoll

Um diese beiden Mengen zu verkleinern, wird weiterhin an neuen Typsystem geforscht.

GHC bietet dazu bereits viele Spracherweiterungen an:
GADTs, *Rank-N Types*, *Type Families*, etc.

Aber es gibt noch mehr, z.B. Dependent Types Agda, Idris, Coq

GADTs: Generalised Algebraic Datatypes

Erweiterung **Generalised Algebraic Datatypes** verallgemeinert:

1. Konstruktoren werden durch Ihren Typ beschrieben
2. *Ergebnistyp* darf *beliebige Instanz* des deklarierten Typen sein

Beispiel

Arithmetische Ausdrücke von Folie 7.47:

```
data Expr = Const Integer
          | Plus  Expr Expr
          | Times Expr Expr
```

könnte man mit Erweiterung GADTs so schreiben:

```
{-# LANGUAGE GADTs #-}
data Expr where
  Const :: Integer       -> Expr
  Plus  :: Expr -> Expr -> Expr
  Times :: Expr -> Expr -> Expr
```

An der Verwendung ändert sich dabei gar nichts.

GADTs: Generalised Algebraic Datatypes

Erweiterung **Generalised Algebraic Datatypes** verallgemeinert:

1. Konstruktoren werden durch Ihren Typ beschrieben
2. *Ergebnistyp* darf *beliebige Instanz* des deklarierten Typen sein

Beispiel

Arithmetische Ausdrücke von Folie 7.47:

```
data Expr = Const Integer
          | Plus  Expr Expr
          | Times Expr Expr
```

```
> :type Plus
Plus :: Expr -> Expr -> Expr
```

könnte man mit Erweiterung GADTs so schreiben:

```
{-# LANGUAGE GADTs #-}
data Expr where
  Const :: Integer       -> Expr
  Plus  :: Expr -> Expr -> Expr
  Times :: Expr -> Expr -> Expr
```

An der Verwendung ändert sich dabei gar nichts.

GADTs: Generalised Algebraic Datatypes

Erweiterung **Generalised Algebraic Datatypes** verallgemeinert:

1. Konstruktoren werden durch Ihren Typ beschrieben
2. *Ergebnistyp* darf *beliebige Instanz* des deklarierten Typen sein

Beispiel

```
{-# LANGUAGE GADTs #-}
data Expr a where
  ConstB :: Bool -> Expr Bool
  ConstI :: Int  -> Expr Int
  Or     :: Expr Bool -> Expr Bool -> Expr Bool
  Add    :: Expr Int  -> Expr Int  -> Expr Int
  IfExpr :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Ermöglicht uns generischen Konstruktor wie hier z.B. `IfExpr`, doch `eval :: Expr a -> a` bleibt *typsicher* definierbar!

Aber: `deriving` nur für ADTs in GADT Syntax möglich

Inferenzprobleme

Spracherweiterungen wie **Rank-N Types** gehen über den Rahmen der Vorlesung heraus. Hier lediglich ein pathologisches Beispiel für Probleme bei der Typinferenz mit Rank-N Types:

```
{-# LANGUAGE RankNTypes #-}
foo :: (forall a . a -> a) -> (b -> b)
foo x = x x

> foo (foo id) 42
42
```

`foo` hat keinen prinzipalen Typ. GHC kann hier keine der beiden akzeptablen Typsignaturen inferieren.

Auch okay, aber nutzlos: `foo :: (forall a . a) -> b`

Rank N-Types sind Typen, bei denen Quantoren für Typvariablen mitten im Typ auftauchen, und nicht nur wie bisher (implizit) vorne.

Typinferenz

Typinferenz berechnet zu jedem Ausdruck den allgemeinsten Typ (bzw. Fehlermeldung, falls der Ausdruck keinen Typ hat).

Beispiel

Der prinzipale Typ einer Funktionsanwendung $e_1\ e_2$ muss aus den prinzipalen Typen von e_1 und e_2 berechnet werden:

$$\frac{\{\} \vdash e_1 :: (\texttt{Int} \to \beta) \to (\gamma \to \beta) \qquad \{x{::}\alpha\} \vdash e_2 :: \alpha \to \texttt{Double}}{\{x{::}\texttt{Int}\} \vdash e_1\ e_2 :: \gamma \to \texttt{Double}}$$

Beide Prämissen müssen mit $[\texttt{Int}/\alpha,\ \texttt{Double}/\beta]$ instanziiert werden, welche die allgemeinste Lösung der Gleichung $\texttt{Int} \to \beta = \alpha \to \texttt{Double}$ ist.

Dazu müssen wir also Typgleichung lösen können.

Unifikation

- Eine Substitution σ ist **allgemeiner** als σ', falls es eine Substitution τ gibt mit $\sigma\tau = \sigma'$.
- Die **speziellere** Substitution σ' ist also eine Instanz von σ.

Zum Lösen von Typgleichungen verwenden wir Unifikation:

- Ein **Unifikator** zweier Typen A und A' ist eine Substitution σ, so dass $A\sigma = A'\sigma$.
- Der **allgemeinste Unifikator** von A und A' ist die allgemeinste Substitution σ, so dass $A\sigma = A'\sigma$.

Beispiel:

$$(\alpha \to \beta) = (\texttt{Int} \to \beta)$$

für diese Typgleichung ist $[\texttt{Int}/\alpha,\ \texttt{Int}/\beta]$ ein Unifikator. Die Substitution $[\texttt{Int}/\alpha]$ ist der allgemeinste Unifikator.

Robinson's Unifikationsalgorithmus

Der Unifikationsalgorithmus unify arbeitet auf einer Menge von Typgleichungen $E = \{A_1 = B_1, \ldots, A_n = B_n\}$ und liefert

- allgemeinste Substitution σ, so dass $A_i\sigma = B_i\sigma$ für alle i gilt
- oder einen Typfehler.

Algorithmus unify:

1. $\text{unify}(\{\}) = \text{id}$ --Fertig
2. $\text{unify}(\{A = A\} \uplus E) = \text{unify}(E)$ --redundante Gleichung
3. $\text{unify}(\{A \to A' = B \to B'\} \uplus E) = \text{unify}(\{A = B, A' = B'\} \uplus E)$
4. $\text{unify}(\{\alpha = B\} \uplus E) = \sigma\,\text{unify}(E\sigma)$ --Subst.-komposition
   mit $\sigma = [B/\alpha]$, falls α in B nicht erwähnt wird.
   Falls α in B vorkommt: Fehlermeldung "Zirkulär"
5. $\text{unify}(\{B = \alpha\} \uplus E) = \text{unify}(\{\alpha = B\} \uplus E)$
6. In allen andern Fällen: Fehlermeldung "Typfehler"
   z.B. wenn $(\texttt{Int} = \texttt{Bool})$ oder $(\texttt{Int} = \alpha \to \beta)$ in E sind.

Beispiele Unifikation

- Gewöhnliches Beispiel:

$$\begin{aligned}
&\text{unify}\{\texttt{Int} \to \beta = \alpha \to \texttt{Double}\} = \text{unify}\{\texttt{Int} = \alpha, \beta = \texttt{Double}\} \\
&= \text{unify}\{\texttt{Int} = \alpha, \beta = \texttt{Double}\} \;= \text{unify}\{\alpha = \texttt{Int}, \beta = \texttt{Double}\} \\
&= [\texttt{Int}/\alpha]\text{unify}\{\beta = \texttt{Double}\} \;= [\texttt{Int}/\alpha][\texttt{Double}/\beta]\text{unify}\{\} \\
&= [\texttt{Int}/\alpha][\texttt{Double}/\beta]\mathit{id} \;= [\texttt{Int}/\alpha, \texttt{Double}/\beta]
\end{aligned}$$

- Nur Variablen:

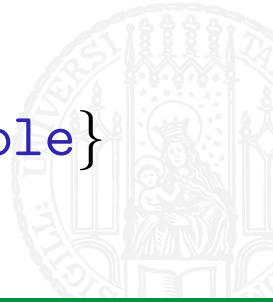
$$\begin{aligned}
&\text{unify}\{\alpha = \beta, \beta = \gamma\} = [\beta/\alpha]\text{unify}\{\beta = \gamma\} = [\beta/\alpha][\gamma/\beta]\text{unify}\{\} \\
&= [\beta/\alpha][\gamma/\beta]\text{id} \;= [\beta/\alpha][\gamma/\beta] \;= [\gamma/\alpha, \gamma/\beta]
\end{aligned}$$

- Typkonflikt:

$$\begin{aligned}
&\text{unify}\{\texttt{Int} \to \alpha = \alpha \to \texttt{Double}\} = \text{unify}\{\texttt{Int} = \alpha,\ \alpha = \texttt{Double}\} \\
&= [\texttt{Int}/\alpha]\text{unify}\{\texttt{Int} = \texttt{Double}\} = \text{"Error: Typfehler"}
\end{aligned}$$

- Zirkulärer Typ:

$$\begin{aligned}
&\text{unify}\{\beta = \beta, \alpha = \alpha \to \texttt{Double}\} = \text{unify}\{\alpha = \alpha \to \texttt{Double}\} \\
&= \text{"Error: Zirkulär"}
\end{aligned}$$



Typinferenzalgorithmus Teil 1

$\text{infer}_\Gamma(e) = (A, \sigma)$ nimmt einen Ausdruck e in Typkontext Γ und liefert den allgemeinsten Typ A von e samt der allgemeinsten Instanziierung σ von Γ, so dass $\Gamma\sigma \vdash e :: A$.

1. $\text{infer}_\Gamma(x) = (A, \text{id})$, falls $x{::}A \in \Gamma$ gilt;
   sonst Fehlermeldung "Ungebundene Variable"

2. $\text{infer}_\Gamma(c) = (A, \text{id})$
   wobei A der passende Typ der Konstanten c ist.

3. $\text{infer}_\Gamma(\backslash x \texttt{ -> } e) = (\alpha\sigma \to B,\ \sigma)$
   wobei wir zuerst für eine frische Typvariable α
   $$\text{infer}_{\Gamma, x::\alpha}(e) = (B, \sigma)$$
   berechnen. Damit gilt $(\Gamma,\ x{::}\alpha)\sigma \vdash e :: B$ und somit ist nach Typregel Abs $\alpha\sigma \to B$ der Typ von $(\backslash x \texttt{ -> } e)$ in Kontext $\Gamma\sigma$.

Typinferenzalgorithmus Teil 2

$\text{infer}_\Gamma(e) = (A, \sigma)$ nimmt einen Ausdruck e in Typkontext Γ und liefert den allgemeinsten Typ A von e samt der allgemeinsten Instanziierung σ von Γ, so dass $\Gamma\sigma \vdash e :: A$.

4. $\text{infer}_\Gamma(e_1\ e_2) = (\,\beta\sigma_3,\ \sigma_1\sigma_2\sigma_3\,)$
   wobei wir zuerst der Reihe nach berechnen:

$$\text{infer}_\Gamma(e_1) = (C, \sigma_1) \quad \text{also} \quad \Gamma\sigma_1 \vdash e_1 :: C$$
$$\text{infer}_{\Gamma\sigma_1}(e_2) = (A, \sigma_2) \quad \text{also} \quad \Gamma\sigma_1\sigma_2 \vdash e_2 :: A$$

   mit dem Lemma zur Typerhaltung unter Substitution (Subst), Folie 8.40, gilt dann auch $\Gamma\sigma_1\sigma_2 \vdash e_1 :: C\sigma_2$
   Für eine frische Typvariable β berechnen wir danach

$$\sigma_3 = \text{unify}\{\ C\sigma_2 = A \to \beta\ \}$$

   Somit gilt erneut nach Subst $\Gamma\sigma_1\sigma_2\sigma_3 \vdash e_1 :: A\sigma_3 \to \beta\sigma_3$
   womit $\beta\sigma_3$ also der Ergebnistyp der Applikation $e_1\ e_2$ ist.

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathrm{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathrm{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ?$ — infer Fall(3)

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathrm{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ?$ — infer Fall(3)

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = ?$ — infer Fall(4)

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathrm{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ?$ — infer Fall(3)

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x) = ?$ — Fall(1)

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathrm{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ?$ — infer Fall(3)

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathrm{id})$ $\mathrm{infer}_{\{y::\alpha, x::\beta\}}(6.9) = ?$ — Fall(2)

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathsf{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ?$ — infer Fall(3)

$\mathsf{infer}_{\{y::\alpha,x::\beta\}}(x\ 6.9)) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha,x::\beta\}}(x) = (\beta, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha,x::\beta\}}(6.9) = (\texttt{Double}, \mathsf{id})$

$\mathsf{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathsf{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ?$ — infer Fall(3)

$\mathsf{infer}_{\{y::\alpha,x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha,x::\beta\}}(x) = (\beta, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha,x::\beta\}}(6.9) = (\texttt{Double}, \mathsf{id})$

$\mathsf{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathsf{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ((\texttt{Double} \to \gamma) \to \gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha, x::\beta\}}(6.9) = (\texttt{Double}, \mathsf{id})$

$\mathsf{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathsf{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ((\texttt{Double} \to \gamma) \to \gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha, x::\beta\}}(6.9) = (\texttt{Double}, \mathsf{id})$

$\mathsf{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

$\mathsf{infer}_{\{y::\alpha\}}(y\ 42) = ?$ — infer Fall(4)

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathrm{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ((\texttt{Double} \to \gamma) \to \gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathrm{id})$ $\quad$ $\mathrm{infer}_{\{y::\alpha, x::\beta\}}(6.9) = (\texttt{Double}, \mathrm{id})$

$\mathrm{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

$\mathrm{infer}_{\{y::\alpha\}}(y\ 42) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha\}}(y) = (\alpha, \mathrm{id})$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathrm{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ((\texttt{Double} \to \gamma) \to \gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathrm{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathrm{id})$ $\quad$ $\mathrm{infer}_{\{y::\alpha, x::\beta\}}(6.9) = (\texttt{Double}, \mathrm{id})$

$\mathrm{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

$\mathrm{infer}_{\{y::\alpha\}}(y\ 42) = ?$ — infer Fall(4)

$\mathrm{infer}_{\{y::\alpha\}}(y) = (\alpha, \mathrm{id})$ $\quad$ $\mathrm{infer}_{\{y::\alpha\}}(42) = (\texttt{Int}, \mathrm{id})$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathsf{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ((\texttt{Double} \to \gamma) \to \gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha, x::\beta\}}(6.9) = (\texttt{Double}, \mathsf{id})$

$\mathsf{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

$\mathsf{infer}_{\{y::\alpha\}}(y\ 42) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha\}}(y) = (\alpha, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha\}}(42) = (\texttt{Int}, \mathsf{id})$

$\mathsf{unify}\{\alpha = \texttt{Int} \to \delta\} = [\texttt{Int} \to \delta/\alpha]$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathsf{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ((\texttt{Double} \to \gamma) \to \gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha, x::\beta\}}(6.9) = (\texttt{Double}, \mathsf{id})$

$\mathsf{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

$\mathsf{infer}_{\{y::\alpha\}}(y\ 42) = (\delta, [\texttt{Int} \to \delta/\alpha])$

$\mathsf{infer}_{\{y::\alpha\}}(y) = (\alpha, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha\}}(42) = (\texttt{Int}, \mathsf{id})$

$\mathsf{unify}\{\alpha = \texttt{Int} \to \delta\} = [\texttt{Int} \to \delta/\alpha]$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathsf{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = ?$ — infer Fall(4)

$\mathsf{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ((\texttt{Double} \to \gamma) \to \gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha, x::\beta\}}(6.9) = (\texttt{Double}, \mathsf{id})$

$\mathsf{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

$\mathsf{infer}_{\{y::\alpha\}}(y\ 42) = (\delta, [\texttt{Int} \to \delta/\alpha])$

$\mathsf{infer}_{\{y::\alpha\}}(y) = (\alpha, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha\}}(42) = (\texttt{Int}, \mathsf{id})$

$\mathsf{unify}\{\alpha = \texttt{Int} \to \delta\} = [\texttt{Int} \to \delta/\alpha]$

$\mathsf{unify}\{((\texttt{Double} \to \gamma) \to \gamma) = \delta \to \eta\} = [(\texttt{Double} \to \gamma)/\delta, \gamma/\eta]$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

$\mathsf{infer}_{\{y::\alpha\}}((\backslash x \text{ -> } x\ 6.9)\ (y\ 42)) = (\gamma, [\texttt{Int} \to (\texttt{Double} \to \gamma)/\alpha, \ldots])$

$\mathsf{infer}_{\{y::\alpha\}}(\backslash x \text{ -> } x\ 6.9) = ((\texttt{Double} \to \gamma) \to \gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x\ 6.9)) = (\gamma, [\texttt{Double} \to \gamma/\beta])$

$\mathsf{infer}_{\{y::\alpha, x::\beta\}}(x) = (\beta, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha, x::\beta\}}(6.9) = (\texttt{Double}, \mathsf{id})$

$\mathsf{unify}\{\beta = \texttt{Double} \to \gamma\} = [\texttt{Double} \to \gamma/\beta]$

$\mathsf{infer}_{\{y::\alpha\}}(y\ 42) = (\delta, [\texttt{Int} \to \delta/\alpha])$

$\mathsf{infer}_{\{y::\alpha\}}(y) = (\alpha, \mathsf{id})$ $\quad$ $\mathsf{infer}_{\{y::\alpha\}}(42) = (\texttt{Int}, \mathsf{id})$

$\mathsf{unify}\{\alpha = \texttt{Int} \to \delta\} = [\texttt{Int} \to \delta/\alpha]$

$\mathsf{unify}\{((\texttt{Double} \to \gamma) \to \gamma) = \delta \to \eta\} = [(\texttt{Double} \to \gamma)/\delta, \gamma/\eta]$

Ergebnis: $\{y::\texttt{Int} \to \texttt{Double} \to \gamma\} \vdash (\backslash x \text{ -> } x\ 6.9)\ (y\ 42)\ ::\ \gamma$

Beispiel

Gegeben t = (\x -> x 6.9) (y 42)

Aufgabe Prinzipalen Typ inferieren

Zuerst vollständig Klammern und freien Variablen frische Typvariablen zuweisen: $\mathrm{FV}(t) = \{y\}$ also setze $\Gamma = \{y::\alpha\}$.

Bemerkung

In diesem Beispiel ist der inferierte Typ γ für Programmausdruck t gar nicht so spannend.

Wichtiger ist dieses Mal der berechnete Typ für die freie Variable y. Über die zurückgelieferte Substitution erkennen wir, dass die freie Variable y den Typ `Int` $\to$ `Double` $\to \gamma$ haben muss.

γ ist dabei frei bleibend, d.h. der vorgegebene Term t schränkt diesen Typ nicht ein.

Ergebnis: $\{y::\texttt{Int} \to \texttt{Double} \to \gamma\} \vdash$ (\x -> x 6.9) (y 42) :: γ

Typinferenzalgorithmus Anmerkungen

- Der beschriebene Typinferenzalgorithmus liefert immer einen prinzipalen Typ, falls der Term überhaupt typisierbar ist.
- Der Algorithmus hat zwar theoretisch eine hohe Komplexität, ist aber in der Praxis sehr schnell durchführbar.
- Der Algorithmus ist auch heute noch die Grundlage für die Typinferenz in GHC und vielen anderen Sprachen.

Der Algorithmus geht auf mehrere Personen zurück:

- Ursprung bei Curry & Feys (1958)
- Erweitert und Allgemeinheit bewiesen bei Hindley (1969)
- Milner zeigte 1979 unabhängig das Gleiche wie Hindley
- Erweitert und Vollständigkeit bewiesen von Damas (1982)

Dementsprechend wird Typinferenz dieser Art als *Hindley-Milner-Damas Typinferenz* bezeichnet.

Zusammenfassung

- Typsysteme weit-verbreitete leichte Form der Spezifikation, welche automatisch gut überprüfbar ist
- Typisierung in Form von Typurteilen: $\Gamma \vdash e::A$
- Kontrolle eines Typurteils durch Typherleitung in Kalkül natürlichen Schließens
- Typherleitungen haben eine Baum-Struktur. Wenn alle Blätter Axiome sind, dann ist die Herleitung ein Beweis.
- Hindley-Milner-Damas Typinferenz berechnet prinzipalen Typ; und beruht auf Robinson's Unifikation.
- Well-typed programs can't go wrong Robin Milner