# Programmierung und Modellierung mit Haskell

## Typsysteme & Typinferenz

Martin Hofmann Steffen Jost

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

2. Juni 2014

# 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 Typrüfung 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 Fehlermeldung nerven beim Kompilieren

# Programmausdrücke

Zur Vereinfachung betrachten wir nur ein Fragment von Haskell:
Der $\lambda$-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 Fkt.-Anwendung.

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`

# Programmausdrücke

Zur Vereinfachung betrachten wir nur ein Fragment von Haskell:
Der $\lambda$-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 Fkt.-Anwendung.

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: $\beta$-Reduktion

$(\backslash x \to e_1)\ e_2 \rightsquigarrow\ e_1[e_2/x]$

# Termsubstitution:

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

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

Bei Substitutionen muss man aufpassen, dass freie Variablen nicht versehentlich eingefangen werden; ggf. müssen gebundene Variablen umbenannt werden: $\backslash x\ \texttt{->}\ e$ ist ja 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



Siehe dazu auch Folie 7-2.

## Formale Definition:

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

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. $\mathrm{FV}\big((\backslash x\ \texttt{->}\ y\ x)\ (\backslash z\ \texttt{->}\ x)\big) = \{x, y\}$ Diese Vorkommen sind getrennt zu betrachten.

# Typausdrücke

Ein **Typausdruck** $A$ is definiert durch:

| | | | | |
|---|---|---|---|---|
| $A, B$ | $::=$ | $\alpha \mid \beta \mid \ldots$ | 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` | :: | `String` | 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** $\Gamma$ 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* |

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

# Typurteil

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

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

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

Formal ist $\_ \vdash \_ :: \_$ eine dreistellige Relation zwischen Typkontexten, Ausdrücken und Typen.

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:

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

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 in der Regel Typurteile
- Typregeln ohne Prämissen heißen **Axiome**

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

# Typregel Var

Der Typ einer Variablen ist durch den Kontext gegeben:

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

Da $\Gamma$ als endliche Abbildung aufgefasst werden kann, können wir diese Typregel auch äquivalent kurz schreiben durch:

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

Beispiele:

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

# Typregel Const

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

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

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

Mehrere Konstanten fassen wir mit folgenden Schemen zusammen:

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

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

# Typregel Funktionsanwendung

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

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

Diese Regel ist nicht anwendbar, falls:

- $e_1$ nicht von einem Funktionstypen ist
- $e_2$ nicht von Typ $A$ ist, d.h. nicht im Definitionsbereich liegt.

Beispiel:

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

# Typregel Funktionsabstraktion

Die Typregel für anonyme Funktionsabstraktion lautet:

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

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

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.

# 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:

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

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 Abs haben wir zur Vereinfachung nur für ein Argument definiert.

# Beispiel Herleitungen

Eine Herleitung in linearer Schreibweise:

| | | |
|---|---|---|
| 1 | $\{x::\alpha,\ f::\alpha \to \beta\} \vdash f::\alpha \to \beta$ | Var |
| 2 | $\{x::\alpha,\ f::\alpha \to \beta\} \vdash x::\alpha$ | Var |
| 3 | $\{x::\alpha,\ f::\alpha \to \beta\} \vdash f\ x::\beta$ | App(1, 2) |
| 4 | $\{x::\alpha\} \vdash \backslash f \to f\ x::(\alpha \to \beta) \to \beta$ | Abs(3) |
| 5 | $\{\} \vdash \backslash x \to (\backslash f \to f\ x)::\alpha \to (\alpha \to \beta) \to \beta$ | Abs(4) |

- Bei dieser Schreibweise müssen wir alle Zeilen nummerieren. Bei jeder Regelanwendung listen wir dann die verwendeten Prämissen der Reihe nach auf.
- Erstellt werden Herleitungen normalerweise von unten nach oben, gelesen werden sie aber meist von oben nach unten.

# 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:

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

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

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

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

# Typregel Konditional

Wir können weitere Programmausdrücke durch neue Typregeln behandeln:

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

## Neue Typregel:

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

## Bedeutung:

- Bedingung muss Typ `Bool` haben
- Zweige 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 Substitutionsmodell durch **Auswerteregeln** formalisieren, wie z.B. Operationale Semantik

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

# Beweisidee:

Solche Beweise werden üblicherweise mit Induktion über die Länge der Typherleitung geführt:
Um zu zeigen, dass $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ 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.

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

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

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

# Curry-Howard-Isomorphismus

Typherleitungen 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

# Programmierung und Modellierung mit Haskell

## Typsysteme & Typinferenz

Martin Hofmann Steffen Jost

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

5. Juni 2014

# Ankündigung

## Aufholübung am 12.06., 17h, B101

Einige unserer engagierten Tutoren haben sich freundlicherweise bereit erklärt, den freien Vorlesungstermin für eine Aufholübung zu nutzen.

Kommen Sie mit Ihren Fragen zu vergangenen Übungsblättern!

Falls Ihnen momentan noch keine Fragen einfallen, Sie aber noch nicht alle bisher gestellten Aufgaben gelöst haben: kommen und lösen Sie die Aufgaben einfach an Ort und Stelle. Sollten dabei dann Fragen auftauchen, so wird Ihnen einer der anwesenden Tutoren Hilfe anbieten, egal ob zu Präsenz- oder Hausübungen.

# Bisher behandelte Typregeln



Folgende Typregeln wurden bisher in der Vorlesung behandelt:

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

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

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

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

# Bisher behandelte Typregeln



Folgende Typregeln wurden bisher in der Vorlesung behandelt:

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

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

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

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

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

# Kontexterweiterung

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

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

**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:

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

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:

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

# Kontraktion

Mehrfachverwendung von Variablen ist harmlos:

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

*Zur Erinnerung:*

$e[z/x, z/y]$ bezeichnet den Term $e$, bei den alle freie Vorkommen von $x$ und $y$ ersetzt werden (siehe Folie 8).

*Merke: $z/x$ bedeutet: $z$ nimmt einen Stock / und erschlägt $x$*

# Strukturelle Typregeln

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

- 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:

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

# 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 $\alpha$ und $\beta$ (**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** $\sigma$ ist eine endliche Abbildung von Typvariablen $\alpha$ 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 $\sigma$ 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 $\sigma$ auf einen Typkontext $\Gamma$ 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:

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

# 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 **Identitäts**substitution $\mathrm{id} = [\,]$. Es gilt $\mathrm{id}\,\sigma = \sigma$ und $\sigma\,\mathrm{id} = \sigma$.

# Prinzipale Typisierung

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

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

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

  Für volles Haskell gilt dies aber nicht mehr.

- 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 ist die Suche nach allgemeinen Typen grundsätzlich eingeschränkt:

```
> :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
```

Option `NoMonomorphismRestriction` erweitert die Suche:

```
> :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 können.

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. Siehe H4-1e

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

## 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 cannot infer this type!
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, wie GADTs, Rank-N Types, Type Families, etc.
Aber es gibt noch mehr, z.B. Dependent Types Agda, Idris, Coq

# 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) -> b      --OK, aber nutzlos
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.

Rank N-Types sind Typen, bei denen Quantoren für Typvariablen mitten im Typ auftauchen, und nicht nur wie bisher ganz 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 $\sigma$ ist **allgemeiner** als $\sigma'$, falls es eine Substitution $\tau$ gibt mit $\sigma\tau = \sigma'$.
- Die **speziellere** Substitution $\sigma'$ ist also eine Instanz von $\sigma$.

Zum Lösen von Typgleichungen verwenden wir Unifikation:

- Ein **Unifikator** zweier Typen $A$ und $A'$ ist eine Substitution $\sigma$, so dass $A\sigma = A'\sigma$.
- Der **allgemeinste Unifikator** von $A$ und $A'$ ist die allgemeinste Substitution $\sigma$, 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 $\sigma$, 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 $\alpha$ in $B$ nicht erwähnt wird.
   Falls $\alpha$ 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 $(\texttt{Int} = \alpha \to \beta)$ $E$ sind.

# Beispiele Unifikation

- Nur Variablen:

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

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

# Typinferenzalgorithmus Teil 2

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

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

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

   wegen Typerhaltung unter Substitution (Subst) gilt dann auch
   $$\Gamma\sigma_1\sigma_2 \vdash e_1 : C\sigma_2$$
   Für eine frische Typvariable $\beta$ berechnen wir danach
   $$\sigma_3 = \mathsf{unify}\{\ C\sigma_2 \ = \ A \to \beta\ \}$$
   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.

# 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