Universität Siegen Fachbereich 12 · Elektrotechnik und Informatik

# Theorie der Programmierung I

**Einführung in die Semantik von Programmiersprachen**

Lukas Weiß Simon Meurer Sebastian Neuser

18. März 2011

Lehrstuhl für Compilerbau und Softwareanalyse
Dozent: Kurt Sieber

# Inhaltsverzeichnis

# 1 Semantik von Programmiersprachen

## 1.1 Was ist Semantik?

Im engeren Sinne: das Laufzeitverhalten von Programmen (dynamische Semantik)

Im weiteren Sinne: alles, was über kontextfreie Syntax hinaus geht, also Kontextbedingungen (statische Semantik), z. B.

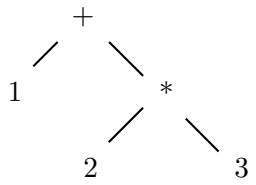
1. jeder Name der benutzt wird, muss zuvor deklariert sein
2. alle Typen müssen „zusammenpassen“, z. B. müssen Anzahl und Typen der formalen und der aktuellen Parameter einer Funktion übereinstimmen.

Die Bedingungen (1) und (2) kann man durch ein Typsystem erklären. (später)

Ziel eines Typsystems: „Grobe“ Programmierfehler sollen zur Compile-Zeit erkannt werden.
$\rightsquigarrow$ keine „Laufzeittypfehler“

Zunächst: Beschreibung des Laufzeitverhaltens einer (kleinen) funktionalen Programmiersprache.

Methode: Syntax „weitgehend ignorieren“, indem man abstrakte Syntax verwendet:
Man nimmt implizit an, dass die syntaktische Struktur eines Programms bereits bekannt ist, d. h. dass das Programm schon als Syntaxbaum vorliegt, z. B. $1 + 2 * 3$ bedeutet:




Hier: Typsysteme und operationelle Semantik am Beispiel einer (idealisierten) Programmiersprache, zunächst: funktional

Die Programmisprache orientiert sich an *ML* (speziell: *O'Caml*). Über die Beschreibung hinaus werden Eigenschaften der Programmiersprache bewiesen, insbesondere Typsicherheit.

Slogan: "Well typed programs don't go wrong."

In unserer Theorie: Wohlgetypte Programme können (zur Laufzeit) nicht steckenbleiben.

# 2 Eine ungetypte funktionelle Sprache (O'Caml ohne Typen)

Wir benutzen abstrakte Syntax, d. h. die KFG ist zu verstehen als Definition einer Menge von Syntaxbäumen (und nicht von Zeichenreihen).

Vorgegeben sind:

- eine unendliche Menge *Id* von Namen *id* (engl.: *identifier*)
- die Menge *Int* aller (Darstellungen von) ganzen Zahlen $n$
- die Menge $Bool = \{true,\ false\}$ der booleschen Werte $b$

Darauf aufbauend: Kontextfreie Syntax der Sprache $\mathcal{L}_1$

Die Menge *Op* aller Operatoren *op* ist definiert durch:

| | | | |
|---|---|---|---|
| *Op* | ::= | $+ \mid - \mid * \mid / \mid \mathbf{mod}$ | arithmetische Operatoren |
| | $\mid$ | $< \mid > \mid \leq \mid \geq \mid =$ | Vergleichsoperatoren |

Die Menge *Const* aller Konstanten *c* (engl.: *constant*) ist definiert durch:

| | | | |
|---|---|---|---|
| *c* | ::= | () | ***unit***-Element |
| | $\mid$ | *b* | boolesche Werte |
| | $\mid$ | *n* | ganze Zahlen |
| | $\mid$ | *op* | Operator |

Die Menge *Exp* aller Ausdrücke *e* (engl.: *expression*) ist definiert durch:

| | | | |
|---|---|---|---|
| *e* | ::= | *c* | Konstante |
| | $\mid$ | *id* | Name |
| | $\mid$ | $e_1\, e_2$ | Applikation |
| | $\mid$ | $\boldsymbol{\lambda} id\boldsymbol{.} e_1$ | $\boldsymbol{\lambda}$-Abstraktion |
| | $\mid$ | $\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ | bedingter Ausdruck |
| | $\mid$ | $\mathbf{let}\ id = e_1\ \mathbf{in}\ e_2$ | **let**-Ausdruck (mit lokaler Deklaration) |

Vergleich zu anderen funktionalen Sprachen:
$\boldsymbol{\lambda} id\boldsymbol{.} e$ schreibt man in *O'Caml*: *fun id* $\rightarrow$ *e*
*SML*: *fn id* $\Rightarrow$ *e*
*Scheme*: *(lambda(id)e)*

Zur Erinnerung:

- $\boldsymbol{\lambda}$-Abstraktionen stehen für namenlose Funktionen, z. B. $\boldsymbol{\lambda} x\boldsymbol{.} + x\, 1$ (Nachfolgerfunktion).
- Applikationen ermöglichen es, Funktionen auf ihre Argumente anzuwenden, bspw. $(\boldsymbol{\lambda} x\boldsymbol{.} + x\, 1)\, 5 \rightarrow 6$.

**Beispiel 1:** $\mathbf{let}\ square = \boldsymbol{\lambda} x\boldsymbol{.} * x\, x\ \mathbf{in}\ square\, (square\, 5)$

**Beispiel 2:** $(\boldsymbol{\lambda} x\boldsymbol{.} * x\, x)\, ((\boldsymbol{\lambda} x\boldsymbol{.} * x\, x)\, 5)$

## 2.1 Etwas konkretere Syntax (für Beispiele)

- Applikation ist linksassoziativ, d. h. $e_1\, e_2\, e_3$ steht für $(e_1\, e_2)\, e_3$
  z. B. $+\, x\, 1$ steht für $(+\, x)\, 1$
- Applikation bindet stärker als alles andere ($\boldsymbol{\lambda}$, **if**, **let**,...), deshalb muss insbesondere jede $\boldsymbol{\lambda}$-Abstraktion, die als Teil einer Applikation auftritt, geklammert werden, z. B. $(\boldsymbol{\lambda} x. + x\, 1)\, 5$

## 2.2 Semantikbeschreibung

Es gibt verschiedene Methoden zur Beschreibung der Semantik:

1. Operationelle Semantik
   Ausdrücke werden „vereinfacht“ oder „umgeformt“, bis man (evtl.) ein Resultat erhält.

2. Denotationelle Semantik
   (Abstraktes) mathematisches Modell (in dem insbesondere $\boldsymbol{\lambda}$-Abstrakionen als Funktionen aufgefasst werden)

3. Axiomatische Semantik
   Axiome und Regeln, mit denen man die Korrektheit (kleiner) Programme beweisen kann

Hier: Nur operationelle Semantik.

### Operationelle Semantik:

Prinzip: Man beschreibt einen Interpreter, d. h. einen Mechanismus (=„abstrakte Maschine“), mit dem man Programme von $L_1$ „ausführen“ kann.

Bei funktionalen Sprachen: „Regeln“ zur Umformung von Ausdrücken. Ein Ausdruck wird solange umgeformt (vereinfacht), bis sich ein Resultat ergibt (oder er bleibt stecken oder gerät in eine endlos-Schleife).

Wichtig: Operationelle Semantik soll verständlich sein, auf Effizienz (des Interpreters) wird keinen Wert gelegt.

Deshalb: Operationelle Semantik orientiert sich an der Struktur der Programme.
$\Rightarrow$ (*sos = structured operational semantics*)

Zwei Möglichkeiten:

- Iterative Beschreibung der Umformungen: **Small step Semantik**: (einzelne Umformungsschritte sind sichtbar)
- Rekursive Beschreibung der Umformungen: **Big step Semantik**: (das Ergebnis wird scheinbar in einem Schritt erreicht)

Zuerst: Small step Semantik

# 3 Small step Semantik

**Beispiel 3:** **let** $square = \boldsymbol{\lambda} x\boldsymbol{.} * x\, x$ **in** $square\, 5 \xrightarrow[\text{Einsetzen der Deklaration}]{} (\boldsymbol{\lambda} x\boldsymbol{.} * x\, x)\, 5$

$\xrightarrow[\text{Parameterübergabe}]{} * 5\, 5 \xrightarrow[\text{Auswertung des Operators}]{} 25$

JETZT: Formal festlegen, welche Umformungsschritte erlaubt sind: Es sollte in jeder Situation höchstens ein Umformungsschritt erlaubt sein.

INTUITION: Werte sind die Ausdrücke, die wir als „ordentliche“ Ergebnisse betrachten.

## 3.1 Small step Semantik von $\mathcal{L}_1$ (und $\mathcal{L}_0$)

Vorgegeben sei eine Menge *Exn* von Ausnahmen (engl.: *exceptions*) *exn*, die „passende“ Elemente wie z. B. *division_by_zero* enthält. Die Menge *EP* der Ausnahmepakete (engl.: *exception packets*) sei definiert durch

$$EP \quad ::= \quad \uparrow exn$$

und für jeden Operator *op* sei eine Funktion $op^{\mathcal{I}}$ vorgegeben mit

$$\begin{array}{ll} op^{\mathcal{I}} : Int \times Int \to Int \cup EP & \text{falls } op \text{ arithmetischer Operator} \\ op^{\mathcal{I}} : Int \times Int \to Bool & \text{falls } op \text{ Vergleichsoperator} \end{array}$$

Die Funktion $op^{\mathcal{I}}$ bezeichnet man als Interpretation des Operators *op*.

Die Menge $Val \subseteq Exp$ der Werte (engl.: *values*) $v$ sei definiert durch

$$v \quad ::= \quad c \;\mid\; \boldsymbol{\lambda} id\boldsymbol{.} e \;\mid\; op\ v_1$$

**Definition 1:** Ein small step (Übergangsschritt, Reduktionsschritt) ist eine „Formel“ der Form $e \to e'$ oder $e \to \uparrow exn$.
Ein small step heißt gültig, wenn er sich mit den folgenden Regeln herleiten lässt.

**Standard-Regeln:**

(OP) $\quad op\ n_1\, n_2 \to op^{\mathcal{I}}(n_1, n_2)$

(BETA-V) $\quad (\boldsymbol{\lambda} id\boldsymbol{.} e)\, v \to e[v/id]$

(APP-LEFT) $\quad \dfrac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2}$

(APP-RIGHT) $\quad \dfrac{e \to e'}{v\, e \to v\, e'}$

(COND-EVAL) $\quad \dfrac{e_0 \to e_0'}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \to \textbf{if } e_0' \textbf{ then } e_1 \textbf{ else } e_2}$

(COND-TRUE) $\quad \textbf{if } true \textbf{ then } e_1 \textbf{ else } e_2 \to e_1$

(COND-FALSE) $\quad \textbf{if } false \textbf{ then } e_1 \textbf{ else } e_2 \to e_2$

(LET-EVAL) $\quad \dfrac{e_1 \to e_1'}{\textbf{let } id = e_1 \textbf{ in } e_2 \to \textbf{let } id = e_1' \textbf{ in } e_2}$

(LET-EXEC) $\quad \textbf{let } id = v \textbf{ in } e \to e[v/id]$

INTUITION: (BETA-V) beschreibt die Parameterübergabe: Wenn der aktuelle Parameter ein Wert $v$ ist, dann darf man ihn für den formalen Parameter *id* der Funktion einsetzen. $e[v/id]$ ist also der Ausdruck der aus $e$ entsteht, indem man $v$ für *id* einsetzt. Dies muss noch genau definiert werden.

(APP-LEFT) besagt: Wenn sich die linke Seite einer Applikation umformen lässt, dann darf man das tun.

(APP-RIGHT) besagt: Wenn die linke Seite einer Applikation schon „fertig" ist und die rechte Seite lässt sich umformen, dann darf man das tun.

**Beispiel 4:** $(\boldsymbol{\lambda} x.\, +\, x\, 1)\, 3 \to +\, 3\, 1$

## 3.2 Beispiel zur small step Semantik (detailiert)

**Beispiel 5:** Sei $e$ der Ausdruck **let** $square = \boldsymbol{\lambda} x.\, *\, x\, x$ **in** $square\, (square\, 5)$

1. Schritt: Da $\boldsymbol{\lambda} x.\, *\, x\, x$ ein Wert ist, ist (LET-EXEC) anwendbar:
$e \to square\, (square\ 5)[\boldsymbol{\lambda} x.\, *\, x\, x/square] \to (\boldsymbol{\lambda} x.\, *\, x\, x)\, ((\boldsymbol{\lambda} x.\, *\, x\, x)\, 5)$

2. Schritt: Wegen (BETA-V) gilt: $(\boldsymbol{\lambda} x.\, *\, x\, x)\, 5 \to (*\, x\, x)[5/x] \to *\, 5\, 5$
Daraus folgt mit (APP-RIGHT): $(\boldsymbol{\lambda} x.\, *\, x\, x)\, ((\boldsymbol{\lambda} x.\, *\, x\, x)\, 5) \to (\boldsymbol{\lambda} x.\, *\, x\, x)\, (*\, 5\, 5)$

3. Schritt: Wegen (OP) gilt: $*\, 5\, 5 \ \to 25$
Mit (APP-RIGHT) folgt daraus: $(\boldsymbol{\lambda} x.\, *\, x\, x)\, (*\, 5\, 5) \to (\boldsymbol{\lambda} x.\, *\, x\, x)\, 25$

4. Schritt: Mit (BETA-V): $(\boldsymbol{\lambda} x.\, *\, x\, x)\ 25 \to *\, 25\, 25$

5. Schritt: Mit (OP): $*\ 25\ 25 \to 625$

KURZ: $e \xrightarrow[\text{(LET-EXEC)}]{} (\boldsymbol{\lambda} x.\, *\, x\, x)\, ((\boldsymbol{\lambda} x.\, *\, x\, x)\, 5) \xrightarrow[\text{(BETA-V)}]{\text{(APP-RIGHT)}} (\boldsymbol{\lambda} x.\, *\, x\, x)\, (*\, 5\, 5)$
$\xrightarrow[\text{(OP)}]{\text{(APP-RIGHT)}} (\boldsymbol{\lambda} x.\, *\, x\, x)\, 25 \xrightarrow[\text{(BETA-V)}]{} *\, 25\, 25 \xrightarrow[\text{(OP)}]{} 625$

## 3.3 Eigenschaften der small step Semantik beweisen

WIR WOLLEN ZEIGEN: Für jeden Ausdruck $e$ gibt es genau eine Berechnung.

SCHREIBWEISE: Wenn kein small step $e \to e'$ oder $e \to \uparrow exn$ existiert, so schreibt man $e \nrightarrow$.

**Lemma 1:** *Für alle $v \in Val$ gilt: $v \nrightarrow$ (Für Werte gibt es keine small steps, d. h. sie sind tatsächlich „fertig ausgewertet".)*

**Beweis:** Wenn $v = c$, *id*, oder $\boldsymbol{\lambda}$-Abstraktion, dann existiert keine small step Regel, deren Konstruktion von der Form $v \to \ldots$ ist.
Wenn $v = op\ v_1$, dann kämen (OP), (BETA-V), (APP-LEFT) oder (APP-RIGHT) in Frage. (OP), (BETA-V) und (APP-LEFT) passen nicht. (APP-RIGHT) geht ebenfalls nicht, da die Prämisse von der Form $op \to \ldots$ oder $v_1 \to \ldots$ sein müsste, was (nach Induktionsannahme: Induktion über die Größe von $v$) nicht möglich ist. □

**Satz 1:** *Die small step Semantik ist deterministisch, d. h. für jeden Ausdruck e existiert höchstens ein small step $e \to e'$ oder $e \to \uparrow exn$.*

**Beweis:** (Exception-Regeln ignoriert!)

IDEE: In jeder Situation ist höchstens eine Regel anwendbar und diese führt zu einem eindeutigen „Resultat".

TECHNIK: Induktion über die Größe von $e$ (wegen den Regeln mit Prämissen) und Fallunterscheidung nach der Form von $e$.

1. Fall $e = c$ oder $e = id$ oder $e = \lambda id.e_1 \rightsquigarrow$ Es existiert kein small step.

2. Fall $e = e_1 e_2 \rightsquigarrow$ Es kommen in Frage: (App-Left), (App-Right), (Beta-V), (Op)

   Fall 2a: $e_1 \to e_1' \rightsquigarrow$ es ist nur (App-Left) anwendbar, denn alle übrigen Regeln verlangen, dass $e_1$ ein Wert ist.

   Fall 2b: $e_1 \in Val$,$e_2 \to e_2' \rightsquigarrow$ Nur (App-Right) anwendbar. Insbesondere sind (Op) und (Beta-V) nicht möglich, da sie als rechte Seite der Applikation einen Wert verlangen. (App-Left) passt nicht wegen $e_1 \not\to$

   Fall 2c: $e_1 \in Val$,$e_2 \in Val \rightsquigarrow$ Höchstens (Op) oder (Beta-V) können angewendet werden, diese schließen sich aber gegenseitig aus.

In allen Fällen muss man sich auch noch überzeugen, dass das Resultat der Anwendung der Regel eindeutig ist. Dabei muss die Induktionsannahme ausgenutzt werden, z. B. im Fall 2a: Da $e_1'$ nach Induktionsannahme durch $e_1$ bestimmt ist, ist auch $e_1' e_2$ eindeutig durch $e_1 e_2$ bestimmt. Weitere Fälle: Übung! □

## 3.4 Exceptions

Idee: Exceptions „schlagen durch“ oder „pflanzen sich nach außen fort“, d. h. wenn bei der Auswertung eines Teilausdrucks gemäß einer der „wo“-Regeln (App-Left), (App-Right), (Cond-Eval), (Let-Eval) eine Exception geworfen wird, dann wird sie auch vom Gesamtausdruck geworfen.

Formal: Zu jeder dieser Regeln formuliert man eine entsprechende exception-Regel.

Exceptions entstehen bisher nur bei (Op), z. B. $/\,1\,0 \xrightarrow[\text{(Op)}]{} \uparrow \mathit{division\_by\_zero}$, da

$/^{\mathcal{I}} : Int \times Int \to Int \cup EP$:

$$/^{\mathcal{I}}(m,n) \begin{cases} \text{ganzzahliger Quotient von m und n,} & \text{falls } n \neq 0 \\ \to \uparrow \mathit{division\_by\_zero} & \text{falls } n = 0 \end{cases}$$

Das lässt sich durch folgede „Meta-Regel“ formulieren: Aus einer „wo“-Regel

$$\text{(R)} \quad \frac{e_1 \to e_1'}{e_2 \to e_2'}$$

entsteht die Exception-Regel

$$\text{(R-Exn)} \quad \frac{e_1 \to ep}{e_2 \to ep}$$

Diese Vorschrift zum Aufstellen von Exception-Regeln gilt auch für zukünftige Spracherweiterungen. Deshalb werden exception-Regeln bei den Spracherweiterungen nicht mehr ausdrücklich angegeben.

**Exception-Regeln:**

(App-Left-Exn) $\dfrac{e_1 \to ep}{e_1\, e_2 \to ep}$

(App-Right-Exn) $\dfrac{e \to ep}{v\, e \to ep}$

(Cond-Eval-Exn) $\dfrac{e_0 \to ep}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \to ep}$

(Let-Eval-Exn) $\dfrac{e_1 \to ep}{\textbf{let } id = e_1 \textbf{ in } e_2 \to ep}$

# 4 Syntaktischer Zucker

Zusätzlich zur (bisher definierten) sogenannten Kernsyntax kann man abkürzende Schreibweisen einführen, die man als syntaktischen Zucker bezeichnet, z. B.:

- infix-Schreibweise für Operatoren: $e_1\ op\ e_2$ für $op\ e_1\ e_2$
- **not** steht für $\boldsymbol{\lambda} x.$**if** $x$ **then** *false* **else** *true*
- $e_1$ && $e_2$ steht für **if** $e_1$ **then** $e_2$ **else** *false*
- $e_1$ || $e_2$ steht für **if** $e_1$ **then** *true* **else** $e_2$
- Funktionsdeklarationen: **let** $id\ id_1\ \ldots\ id_n = e_0$ **in** $e_1$ steht für **let** $= \boldsymbol{\lambda} id_1.\ \ldots\ \boldsymbol{\lambda} id_n.e_0$ **in** $e_1$

Warum nicht gleich diesen Schreibweise?
Die Kernsyntax ist besser für die Theorie (small step, Typsystem) geeignet. ⇒ Beweise lassen sich mit relativ wenigen Fallunterscheidungen führen. Syntaktischer Zucker wird in den Beweisen ignoriert, da nur abkürzende Schreibweise.

Insbesondere: In der Funktionsdeklaration sind zwei Konzepte (Deklaration und $\boldsymbol{\lambda}$-Abstraktion) schon vermischt, die man in der Kernsyntax auseinanderhält.

Prizipiell gilt: Ausdrücke können immer zunächst in die Kernsyntax übersetzt und dann ausgeführt werden.

Alternative: Syntaktischen Zucker direkt ausführen mit sogenannten abgeleiteten Regeln (engl.: *derived rules*).

Idee: Eine abgeleitete Regel soll auf dem syntaktischen Zucker so arbeiten, wie die ursprünglichen Regeln auf der Übersetzung des syntaktischen Zuckers. (⇒ Man kann sich die Übersetzung sparen.)

**Beispiel 6:**

$$\begin{array}{ccc} e_1 \mathbin{\&\&} e_2 & \xrightarrow[\text{(And-Eval)}]{} & e_1' \mathbin{\&\&} e_2 \\ \text{synt. Zucker} \downarrow & & \downarrow \text{synt. Zucker} \\ \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } \textit{false} & \xrightarrow[\text{(Cond-Eval)}]{} & \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } \textit{false} \end{array}$$

$$\begin{array}{ccc} \textit{true} \mathbin{\&\&} e & \xrightarrow[\text{(And-True)}]{} & e \\ \text{synt. Zucker} \downarrow & & \\ \textbf{if } \textit{true} \textbf{ then } e \textbf{ else } \textit{false} & \xrightarrow[\text{(Cond-True)}]{} & e \end{array}$$

$$\begin{array}{ccc} \textit{false} \mathbin{\&\&} e & \xrightarrow[\text{(And-False)}]{} & \textit{false} \\ \text{synt. Zucker} \downarrow & & \\ \textbf{if } \textit{false} \textbf{ then } e \textbf{ else } \textit{false} & \xrightarrow[\text{(Cond-False)}]{} & \textit{false} \end{array}$$

## 4.1 Abgeleitete Regeln

(And-Eval) $\dfrac{e_1 \to e_1'}{e_1 \;\&\&\; e_2 \to e_1' \;\&\&\; e_2}$

(And-True) $\mathit{true} \;\&\&\; e \to e$

(And-False) $\mathit{false} \;\&\&\; e \to \mathit{false}$

(Or-Eval) $\dfrac{e_1 \to e_1'}{e_1 \;||\; e_2 \to e_1' \;||\; e_2}$

(Or-True) $\mathit{true} \;||\; e \to \mathit{true}$

(Or-False) $\mathit{false} \;||\; e \to e$

## 4.2 Infix-Notation

syntaktischer Zucker: $e_1\ op\ e_2$ statt $op\ e_1\ e_2$

Für Infix-Notation muss man Prioritäten für Operatoren einführen:

1. Applikation bindet am stärksten
2. $*, /, mod$
3. $+, -$
4. $=, <=, >=, <, >$
5. $\&\&, ||$

Alle Infix-Schreibweisen sind links-assoziativ, z. B. $e_1 - e_2 - e_3$ steht für $(e_1 - e_2) - e_3$.

*In* TPML *oder* O'Caml*: Wenn ein Operator in der Präfix-Schreibweise benutzt wird, muss er geklammert werden, z. B.* $(+)\,1\,2$.

## 4.3 Funktionsdeklarationen

syntaktischer Zucker: $\mathbf{let}\ id\ id_1\ \ldots\ id_n\ =\ e\ \mathbf{in}\ \ldots$ statt $\mathbf{let}\ id\ =\ \boldsymbol{\lambda} id_1\ \ldots\ \boldsymbol{\lambda} id_n.e\ \mathbf{in}\ \ldots$

**Beispiel 7:** $\mathbf{let}\ square\ x = x * x\ \mathbf{in}\ \ldots$ statt $\mathbf{let}\ square = \boldsymbol{\lambda} x.\ *\,x\,x\ \mathbf{in}\ \ldots$
$\mathbf{let}\ square\_sum\ x\ y = x*x+y*y\ \mathbf{in}\ \ldots$ statt $\mathbf{let}\ square\_sum = \boldsymbol{\lambda} x.\boldsymbol{\lambda} y.+(*\,x\,x)\,(*\,y\,y)\ \mathbf{in}\ \ldots$

## 4.4 Weitere Verkleinerung der Kernsyntax

$\mathbf{let}\ id = e_1\ \mathbf{in}\ e_2$ kann als syntaktischer Zucker für $(\boldsymbol{\lambda} id.e_2)\,e_1$ aufgefasst werden.
Dann sind (Let-Eval) und (Let-Exec) abgeleitete Regeln:

$$\begin{array}{ccc} \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 & \xrightarrow[\text{(Let-Eval)}]{} & \mathbf{let}\ id = e_1'\ \mathbf{in}\ e_2 \\ \text{synt. Zucker}\ \downarrow & & \downarrow\ \text{synt. Zucker} \\ (\boldsymbol{\lambda} id.e_2)\,e_1 & \xrightarrow[\text{(App-Right)}]{} & (\boldsymbol{\lambda} id.e_2)\,e_1' \end{array}$$

$$\begin{array}{ccc} \mathbf{let}\ id = v\ \mathbf{in}\ e & \xrightarrow[\text{(Let-Exec)}]{} & e[v/id] \\ \text{synt. Zucker}\ \downarrow & & \downarrow\ \text{synt. Zucker} \\ (\boldsymbol{\lambda} id.e)v & \xrightarrow[\text{(Beta-V)}]{} & e[v/id] \end{array}$$

# 5 Begriffe zur small step Semantik

Eine Berechnungsfolge ist eine (endliche oder unendeliche) Folge von gültigen small steps, z. B. $e_0 \to e_1 \to \cdots \to e_i$.

Eine Berechnung (für den Ausdruck $e$) ist eine maximale Berechnungsfolge, die mit $e$ beginnt, d. h.: Sie ist entweder endlich und lässt sich nicht weiter fortsetzen oder sie ist unendlich.
Wenn sie unendlich ist, so sagt man, die Berechnung divergiert.
Wenn sie endlich ist und mit einem Wert $v$ oder einer Exception $exn$ endet, so sagt man, sie terminiert.
Wenn sie endlich ist und mit einem Ausdruck $e' \notin \mathit{Val}$ endet, dann sagt man, sie bleibt stecken.

**Beispiel 8:** *(für Divergenz)*
$(\boldsymbol{\lambda} x.x\,x)(\boldsymbol{\lambda} x.x\,x) \xrightarrow[\text{(Beta-V)}]{} xx[\boldsymbol{\lambda} x.x\,x/x] \to (\boldsymbol{\lambda} x.x\,x)(\boldsymbol{\lambda} x.x\,x) \xrightarrow[\text{(Beta-V)}]{} \ldots$

**Beispiel 9:** *(für Steckenbleiben)*

- 1 *true*
- **if** 1 **then** 2 **else** 3
- 1 2 (Applikation)

Ein Thema der Vorlesung: Es sollen Typsysteme eingeführt werden, die solche Ausdrücke schon zur Compile-Zeit ausschließen. Eine Programmiersprache heißt typsicher, wenn das Typsystem zur Compile-Zeit verhindert, dass Ausdrücke zur Laufzeit steckenbleiben.

Zunächst: Weiter mit der ungetypten Sprache.

Noch zu definieren: Substitutionsschreibweise $e[v/id]$ oder allgemeiner $e[e'/id]$

Zur Intuition betrachten wir ein Beispiel zur Parameterübergabe:

**Beispiel 10:** $(\boldsymbol{\lambda} x.*\,x\,x)\,3 \to *\,3\,3$

Idee: $e'$ wird für alle Vorkommen von $id$ in $e$ eingesetzt.

„Für alle Vorkommen“ stimmt nicht ganz, besser: Für alle Vorkommen, die im Gültigkeitsbereich dieses $\boldsymbol{\lambda} id$ liegen. Problem: Dieser Gültigkeitsbereich kann ein „Loch“ haben, nämlich überall dort, wo ein neues „lokales“ $id$ eingeführt wird.

**Beispiel 11:** $(\boldsymbol{\lambda} x.\underbrace{(\boldsymbol{\lambda} x.x+x)}_{\text{„Loch“}}(x+1))\,5 \to (\boldsymbol{\lambda} x.x+x)\,(5+1)$

**Definition 2:** *(naive Substitution)*
Der Ausdruck $e[e'/id]$, der durch naive Substitution von $e'$ für $id$ in $e$ entsteht, wird durch folgende Induktion über die Struktur von $e$ definiert:

1. $c[e'/id] = c$
2. $id'[e'/id] = \begin{cases} e', & \text{falls } id = id' \\ id', & \text{sonst} \end{cases}$
3. $(\boldsymbol{\lambda} id'.e_1)[e'/id] = \begin{cases} \boldsymbol{\lambda} id'.e_1, & \text{falls } id = id' \\ \boldsymbol{\lambda} id'.e_1[e'/id], & \text{sonst} \end{cases}$
4. $(e_1\,e_2)[e'/id] = (e_1[e'/id])\,(e_2[e'/id])$
5. $(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2)[e'/id] = \textbf{if } e_0[e'/id] \textbf{ then } e_1[e'/id] \textbf{ else } e_2[e'/id]$

Da **let** als syntaktischer Zucker aufgefasst werden kann, kann man die Substitution auf **let** übertragen:

$$(\textbf{let } id' = e_1 \textbf{ in } e_2)[e'/id] = \begin{cases} \textbf{let } id' = e_1[e'/id] \textbf{ in } e_2, & \text{falls } id = id' \\ \textbf{let } id' = e_1[e'/id] \textbf{ in } e_2[e'/id], & \text{sonst} \end{cases}$$

**Beispiel 12:**

$$\begin{aligned} (\boldsymbol{\lambda} x.(\boldsymbol{\lambda} x.x + x)\,(x+1))\,5 &\xrightarrow[\text{(Beta-V)}]{} ((\boldsymbol{\lambda} x.x + x)\,(x+1))[5/x] \\ &\to (\boldsymbol{\lambda} x.x + x)[5/x]\,(x+1)[5/x] \\ &\to (\boldsymbol{\lambda} x.x + x)\,(5+1) \\ &\to \ldots \end{aligned}$$

Problem bei der naiven Substitution: Wenn der Ausdruck e' (der eingesetzt werden soll) selbst frei vorkommende Namen enthält, dann kann eine sog. Namenskollision auftreten, d. h. der frei vorkommende Name kann sich „eine neue Bindung einfangen“.

**Beispiel 13:** $(\boldsymbol{\lambda} y.\boldsymbol{\lambda} x.x + y)\,(x+1) \xrightarrow[\text{(Beta-V)}]{} (\boldsymbol{\lambda} x.x + y)[x+1/y] \xrightarrow[\text{naive Subst.}]{} \boldsymbol{\lambda} x.x + x + 1$

Aus einer Funktion die $x + 1$ auf den übergebenen Parameter addiert wird die Funktion, die zu jedem Parameter $x$ das Ergebnis $2x + 1$ liefert.

Lösung: Man führt vor dem eigentlichen Ersetzen eine gebundene Umbenennung des Parameters durch, der zur Namenskollision führt.

Im Beispiel: $(\boldsymbol{\lambda} x.x + y)$ wird zunächst zu $(\boldsymbol{\lambda} x.x' + y)$, dann naive Substitution.

**Definition 3:** Die Menge $free(e)$ aller im Ausdruck $e$ frei vorkommenden Namen ist durch Induktion über die Größe von $e$ definiert:

$free(c) = \emptyset$

$free(id) = id$

$free(e_1\, e_2) = free(e_1) \cup free(e_2)$

$free(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) = free(e_0) \cup free(e_1) \cup free(e_2)$

$free(\boldsymbol{\lambda} id.e_1) = free(e_1) \backslash \{id\}$

$free(\textbf{let } id = e_1 \textbf{ in } e_2) = (free(e_2) \backslash \{id\}) \cup free(e_1)$

Intuition: Ein Vorkommen eines Namens $id$ in $e$, das nicht im Gültigkeitsbereich eines Parameters $\boldsymbol{\lambda} id$ oder einer Definition **let** $id$ liegt, nennt man freies Vorkommen.

**Beispiel 14:** $free((\boldsymbol{\lambda} x.\boldsymbol{\lambda} y.x + y)(y+1)) = free(\boldsymbol{\lambda} x.\boldsymbol{\lambda} y.x + y) \cup free(y+1) = y$

**Definition 4:** $e$ heißt abgeschlossen, wenn $free(e) = \emptyset$, andere Ausdrücke heißen offen.
Wir betrachten (meistens) nur abgeschlossene Ausdrücke.

Mit Hilfe von *free*() kann man jetzt die endgültige Definition der Substitution angeben:

**Definition 5:** *(Substitution, endgültige Version)*
Die Definition von $e[e'/id]$ ist ähnlich wie die naive Definition, der einzige Unterschied liegt im Fall der $\boldsymbol{\lambda}$-Abstraktion:

$$(\boldsymbol{\lambda} id'.e_1)[e'/id] = \begin{cases} \boldsymbol{\lambda} id'.e_1, & \text{falls } id = id' \\ \boldsymbol{\lambda} id''.(e_1[id''/id'][e'/id]), & \text{sonst} \end{cases}$$

wobei $id''$ ein „neuer Name“ ist, d. h. $id'' \notin free(e') \cup \{id\} \cup free(\boldsymbol{\lambda} id'.e_1)$

Analog für **let**.

Das Ersetzen eines Parameters $id'$ durch einen neuen Namen $id''$ bezeichnet man als $\alpha$-Konversion oder gebundene Umbenennung.

Bemerkungen:

1. Wenn $id' \notin free(e')$, d. h. wenn keine Namenskollision auftritt, dann kann man $id'' = id'$ wählen (denn $id' \neq id$ und $id' \notin free(\boldsymbol{\lambda} id' - e_1)$ ist sowieso erfüllt), d. h. in diesem Fall braucht man nicht umzubenennen.
2. Um die Substitutionsschreibweise eindeutig zu machen, kann man eine Vorschrift angeben, mit der der neue Name ausgewählt wird.
3. Wenn $e'$ abgeschlossen ist, dann tritt nie eine Namenskollision auf, also stimmt die neue Definition mit der naiven Substitution überein.

**Lemma 2:**

1. *Wenn $e \to e'$, dann gilt $free(e') \subseteq free(e)$, d. h. es entstehen keine neuen freien Namen durch einen small step.*
2. *speziell: Wenn $e \to e'$ und $e$ abgeschlossen ist, dann ist auch $e'$ abgeschlossen.*

Folgerung: *Wenn wir nur abgeschlossene Ausdrücke betrachten, dann werden auch stets nur solche als Parameter übergeben. In diesem Fall stimmen beide Definitionen der Substitution überein, d. h. man kommt auch mit der naiven Substitution aus.*

**Beispiel 15:** *(für die neue Definition der Substitution)*

$$(\boldsymbol{\lambda} x.\boldsymbol{\lambda} y. + x\, y)\, y \xrightarrow[\text{(Beta-V)}]{} \boldsymbol{\lambda} y. + x\, y)[y/x] \xrightarrow[\text{Substitution}]{} \boldsymbol{\lambda} y'. + y\, y'$$

# 6 Spracherweiterung: Rekursion ($\mathcal{L}_2$)

BISHER: **let** $fact = \boldsymbol{\lambda} x.$**if** $x = 0$ **then** $1$ **else** $x * fact\,(x-1)$ **in** $fact\,3$
syntaktisch erlaubt, liefert aber keine rekursive Funktion.

IN ANDEREN SPRACHEN: Deklaration rekursiver Funktionen möglich, z. B. in O'caml:
**let rec** $f = \boldsymbol{\lambda} x. \ldots$

Wir wollen (nach wie vor) die Konzepte voneinander trennen.

BEREITS EINGEFÜHRTE KONZEPTE:

- Konzept Funktion (mit Parameterübergabe) $\rightsquigarrow$ $\boldsymbol{\lambda}$-Abstraktion
- Konzept Deklaration $\rightsquigarrow$ **let**-Ausdrücke

JETZT: Konzept Rekursion $\rightsquigarrow$ **rec**-Ausdrücke

FORMAL: Die Syntax von $\mathcal{L}_1$ wird erweitert durch

$$e \quad ::= \quad \textbf{rec}\ id.e$$

Die erweiterte Sprache heißt $\mathcal{L}_2$.

INTUITION: **rec** $id.e$ bezeichnet das Element, das durch die Gleichung $id = e$ definiert ist, z. B. **rec.***fact* $\boldsymbol{\lambda} x.$**if** $x = 0$ **then** $1$ **else** $x * fact\,(x-1)$ bezeichnet die Funktion, die durch die Gleichung $fact = \boldsymbol{\lambda} x.$**if** $x = 0 \ldots$ gegeben ist.

Typischerweise wird **rec** $id.e$ nur für $\boldsymbol{\lambda}$-Abstraktionen $e$ benutzt (weil man eine Funktion rekursiv deklarieren will), erlaubt ist aber jeder Ausdruck $e$, z. B. **rec** $x.x + 1$.

## 6.1 Syntaktischer Zucker für Rekursion

**rec**-Ausdrücke gibt es so nicht in Programmiersprachen. Man verbindet sie stets mit einer Deklaration wie folgt:

**let rec** $id = e_1$ **in** $e_2$
steht für
**let** $id =$ **rec** $id\ e_1$ **in** $e_2$
} man spart die Wdh. des Namens $id$

Wie bei der nicht rekursiven Funktionsdeklaration ist auch ist auch hier die traditionelle Parameterschreibweise erlaubt (anstelle der $\boldsymbol{\lambda}$-Abstraktion):

**let rec** $id\ id_1 \ldots id_n = e_1$ **in** $e_2$ steht für **let rec** $id = \boldsymbol{\lambda} id_1 \ldots \boldsymbol{\lambda} id_n.e_1$ **in** $e_2$

**Beispiel 16:** Der Ausdruck
**let rec** $fact\ x =$ **if** $x = 0$ **then** $1$ **else** $\ldots$
steht für
**let rec** $fact = \boldsymbol{\lambda} x.$**if** $\ldots$
was wiederum für
**let** $fact =$ **rec** $fact.\boldsymbol{\lambda} x.$**if** $\ldots$
steht.

## 6.2 Small step Semantik: (Unfold)-Regel

Es wird eine neue Regel hinzugefügt, nämlich eine Regel zum „Auffalten“ der Rekursion:

(UNFOLD) $\quad \textbf{rec}\ id.e \longrightarrow e[\textbf{rec}\ id.e/id]$

BEMERKUNG: Die Menge *Val* der Werte $v$ ist wie vorher definiert, d. h. **rec**-Ausdrücke sind <u>keine</u> Werte.

INTUITION: In die rechte Seite der Gleichung $id = e$ wird die „gesamte rekursive Funktion“ für jedes (freie) Vorkommen von $id$ eingesetzt.

**Beispiel 17:** $\textbf{let rec}\ fact\ x = \textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * fact\,(x-1)\ \textbf{in}\ fact\,1$

$$
\begin{array}{rl}
\text{steht für} & \textbf{let}\ fact = \textbf{rec}\ fact.\boldsymbol{\lambda} x.\textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * fact\,(x-1)\ \textbf{in}\ fact\,1 \\
\xrightarrow[\text{(LET-EVAL)}]{\text{(UNFOLD)}} & \textbf{let}\ fact = \boldsymbol{\lambda} x.\textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * (\textbf{rec}\ldots)\,(x-1)\ \textbf{in}\ fact\,1 \\
\xrightarrow[\text{(LET-EXEC)}]{} & (\boldsymbol{\lambda} x.\textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * (\textbf{rec}\ldots)\,(x-1))\,1 \\
\xrightarrow[\text{(BETA-V)}]{} & \textbf{if}\ 1 = 0\ \textbf{then}\ 1\ \textbf{else}\ 1 * (\textbf{rec}\ldots)\,(1-1) \\
\xrightarrow[(Op)]{} & \textbf{if}\ \mathit{false}\ \textbf{then}\ 1\ \textbf{else}\ 1 * (\textbf{rec}\ldots)\,(1-1) \\
\xrightarrow[\text{(COND-FALSE)}]{} & 1 * (\textbf{rec}\ldots)\,(1-1) \\
\xrightarrow[\text{(UNFOLD)}]{} & 1 * (\boldsymbol{\lambda} x.\textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * (\textbf{rec}\ldots)\,(x-1))\,(1-1) \\
\xrightarrow[(Op)]{} & 1 * (\boldsymbol{\lambda} x.\textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * (\textbf{rec}\ldots)\,(x-1))\,0 \\
\xrightarrow[\text{(BETA-V)}]{} & 1 * (\textbf{if}\ 0 = 0\ \textbf{then}\ 1\ \textbf{else}\ 0 * (\textbf{rec}\ldots)\,(0-1)) \\
\xrightarrow[(Op)]{} & 1 * (\textbf{if}\ \mathit{true}\ \textbf{then}\ 1\ \textbf{else}\ldots) \\
\xrightarrow[\text{(COND-TRUE)}]{} & 1 * 1 \\
\xrightarrow[(Op)]{} & 1
\end{array}
$$

Damit (UNFOLD) stets wohldefiniert ist, müssen die Definitionen von *free*() und der Substitution noch auf die **rec**-Ausdrücke verallgemeinert werden:

$$free(\textbf{rec}\ id.e) = free(e)\backslash\{id\}$$

$$(\textbf{rec}\ id'.e)[e'/id] = \begin{cases} \textbf{rec}\ id'.e, & \text{falls } id = id' \\ \textbf{rec}\ id''.(e[id''/id'][e'/id]), & \text{sonst}^1 \end{cases}$$

BEACHTE: **rec** $id.$ ist ein Bedingungsmechanismus wie $\boldsymbol{\lambda} id.$ und wird deshalb bei *free*() und bei der Substitution behandelt.

## 6.3 Gelten die bisherigen Sätze auch für $\mathcal{L}_2$?

**Lemma 3:** *$v \not\to$ gilt nach wie vor, denn die neue small step Regel (UNFOLD) ist nur auf **rec**-Ausdrücke anwendbar, die <u>nicht</u> zu den Werten zählen.*

**Satz 2:** *(Small steps sind deterministrisch)*
*Für jeden Ausdruck e existiert <u>höchstens</u> ein small step $e \to e'$ oder $e \to \uparrow \mathrm{exn}$.*

**Beweis:** Gilt nach wie vor, denn die Regel (UNFOLD) kann sich mit keiner der alten Regeln überschneiden, da sie nur auf **rec**-Ausdrücke anwendbar ist, auf die die alten Regeln <u>nicht</u> anwendbar sind. Außerdem liefert (UNFOLD) ein eindeutiges Ergebnis. □

[1] wobei $id''$ wieder ein „neuer“ Name ist (gebundene Umbenennung)

**Lemma 4:** *(Bei einem small step entstehen keine neuen freien Namen)*
*Wenn $e \to e'$, dann $free(e') \subseteq free(e)$. Das gilt auch für small steps mit* (Unfold).

**Beweis:** (nur noch für (Unfold) zu zeigen)

$$\begin{array}{ll} & free(e[\textbf{rec}\ id.e/id]) \subset (free(e)\backslash\{id\} \cup free(\textbf{rec}\ id.e)) \\ \underset{\text{per Def. } free()}{=} & (free(e)\backslash\{id\}) \cup (free(e)\backslash\{id\}) \\ = & free(e)\backslash\{id\} \\ \underset{\text{per Def. } free()}{=} & free(\textbf{rec}\ id.e) \end{array}$$

□

## 6.4 Vergleich zur Literatur

Sonst meistens: Fixpunkt-Operator
Es wird nur eine einzige neue Konstante eingeführt, nämlich: $c ::= fix$.
Dann kann man **rec** $id.e$ als synt. Zucker einführen: **rec** $id.e$ steht für $fix(\boldsymbol{\lambda} id.e)$
$\Rightarrow$ *free*() und Substitution müssen nicht erweitert werden, denn beide sind für Konstanten definiert und $fix$ zählt zu den Konstanten.

Mit anderen Worten: Der neue Bindungsmechanismus **rec** $id.e$ wird auf den bereits bekannten $\boldsymbol{\lambda} id.e$ zurückgeführt.

Die (Unfold)-Regel lautet dann:

(Unfold') $\quad fix(\boldsymbol{\lambda} id.e) \to e[fix(\boldsymbol{\lambda} id.e)/id]$

Eine schönere Auffaltungsregel erhält man in *call-by-name* Sprachen, nämlich:

(Unfold") $\quad fix\ e \ \to \ e(fix\ e)$

In Zusammengang mit *call-by-value* ist diese Regel unbrauchbar, denn **rec** $id.e$ steht für

$$\begin{array}{ll} & fix(\boldsymbol{\lambda} id.e) \\ \xrightarrow[\text{(Unfold")}]{} & (\boldsymbol{\lambda} id.e)(fix(\boldsymbol{\lambda} id.e)) \\ \xrightarrow[\text{(Unfold")}]{\text{(App-Right)}} & (\boldsymbol{\lambda} id.e)((\boldsymbol{\lambda} id.e)(fix(\boldsymbol{\lambda} id.e))) \end{array}$$

und führt stets zur Divergenz.

Woher „Fixpunkt“?

Idee: (aus der denotationellen Semantik)
Eine Rekursion liest man als Gleichung für Funktionen, für die man eine Lösung sucht, vgl. Gleichungen der Algebra: $x \ = \ x^2 - 1$
Die Lösung der Gleichung ist nichts Anderes als ein Fixpunkt der Funktion
$F: \ \mathbb{R} \to \mathbb{R}\ x \mapsto x^2 - 1$
Analogie: $fact = \boldsymbol{\lambda} x.\textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * fact(x-1)$
Lösung der Gleichung ist nichts Anderes als Fixpunkt der Funktion
$F : (\mathbb{Z} \hookrightarrow \mathbb{Z}) \to (\mathbb{Z} \hookrightarrow \mathbb{Z})\ f \mapsto \boldsymbol{\lambda} x.\textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * f(x-1)$

# 7 Big step Semantik

IDEE: Die einzelnen small steps interessieren eigentlich nicht. Wichtig ist das Endergebnis einer Berechnung (falls es existiert). In big step Semantik definiert man das Endergebnis *direkt*, d. h. mittels Rekursion statt Iteration.

**Definition 6:** Ein big step ist von der Form $e \Downarrow v$ oder $e \Downarrow exn$

INTUITION: $v$ bzw. $exn$ ist das Endergebnis von $e$.

## 7.1 Big step Semantik von $\mathcal{L}_2$ (und $\mathcal{L}_0$,$\mathcal{L}_1$)

Ein big step ist eine „Formel“ der Gestalt $e \Downarrow r$ mit $r \in Val \cup EP$. Ein big step heißt gültig, wenn er sich mit den Regeln

(VAL) $$v \Downarrow v$$

(OP) $$op\ n_1\ n_2 \Downarrow op^{\mathcal{I}}(n_1, n_2)$$

(BETA-V) $$\frac{e[v/id] \Downarrow v'}{(\boldsymbol{\lambda} id.e)\ v \Downarrow v'}$$

(APP) $$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1\ v_2 \Downarrow v}{e_1\ e_2 \Downarrow v}$$

(COND-TRUE) $$\frac{e_0 \Downarrow true \quad e_1 \Downarrow v}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v}$$

(COND-FALSE) $$\frac{e_0 \Downarrow false \quad e_2 \Downarrow v}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v}$$

(LET) $$\frac{e_1 \Downarrow v_1 \quad e_2[v_1/id] \Downarrow v_2}{\textbf{let } id = e_1 \textbf{ in } e_2 \Downarrow v_2}$$

(UNFOLD) $$\frac{e[\textbf{rec } id.e/id] \Downarrow v}{\textbf{rec } id.e \Downarrow v}$$

und mit den zugehörigen exception-Regeln herleiten lässt. Diese exception-Regeln erhält man aus den obigen Regeln durch folgende Vorschrift: Zu jeder Regel der Form

(R) $$\frac{e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n}{e \Downarrow v}$$

(d. h. zu jeder Regel mit $n$ Prämissen) bildet man für jedes $i \in \{1, \ldots, n\}$ die Regel

(R-EXN-$i$) $$\frac{e_1 \Downarrow v_1 \quad \ldots \quad e_{i-1} \Downarrow v_{i-1} \quad e_i \Downarrow ep}{e \Downarrow ep}$$

Nach dieser Vorschrift entstehen z. B. aus Regel (APP) die drei Regeln

(APP-EXN-1) $$\frac{e_1 \Downarrow ep}{e_1\ e_2 \Downarrow ep}$$

(APP-EXN-2) $$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow ep}{e_1\ e_2 \Downarrow ep}$$

(APP-EXN-3) $$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1\ v_2 \Downarrow ep}{e_1\ e_2 \Downarrow ep}$$

$\xrightarrow{*}$ steht für eine endliche Folge von small steps.
$e \xrightarrow{*} e'$ bedeutet: $\exists e_0, \ldots, e_n (n \geq e') : e = e_0 \to e_1 \to \cdots \to e_n = e'$

**Satz 3:** *(Äquivalenz zwischen small step und big step Semantik)*
*Für alle $e \in Exp$ und alle $r \in Val \cup Exn$ gilt:*

$$e \xrightarrow{*} r \iff e \Downarrow r$$

**Beweis:** Wir zeigen nur: $e \xrightarrow{*} v \iff e \Downarrow v$ (d. h. Exceptions werden ignoriert)

„$\Leftarrow$“: Es gelte $e \Downarrow v$. Zu zeigen: $e \xrightarrow{*} v$

IDEE: Der big step $e \Downarrow v$ ergibt sich aus „kleineren“ big steps (nämlich denen in der Prämisse der ansprechenden Regel).
Für die kleineren big steps existieren nach Indunktionsannahme Folgen von small steps. Diese small step-Folgen setzt man geschickt zu einer Folge zusammen, die dem big step $e \Downarrow v$ entspricht.

BEWEISTECHNIK: Induktion über Länge der Herleitung des big steps $e \Downarrow v$ und Fallunterscheidung nach der zuletzt angewandten big step Regel.

- (VAL) Der big step hat die Form $v \Downarrow v$.
  Es gilt $v \xrightarrow{*} v$ (in 0 Schritten).
- (OP) big step hat die Form $op\ n_1\ n_2 \Downarrow op^3(n_q, n_2)$.
  Es gilt $op\ n_1\ n_2 \xrightarrow{*} op^3(n_1, n_2)$ (in einem Schritt).
- (APP) Es gelte $e_1\ e_2 \Downarrow v$ wegen $e_1 \Downarrow v_1$, $e_2 \Downarrow v_2$ und $v_1\ v_2 \Downarrow v$.
  Nach Induktionsannahme gilt $e_1 \xrightarrow{*} v_1$, $e_2 \xrightarrow{*} v_2$, $v_1\ v_2 \xrightarrow{*} v$.
  Daraus ergibt sich:
  $e_1\ e_2 \underbrace{\xrightarrow[\text{(APP-LEFT)}]{*}}_{\text{auf jede } e_1 \xrightarrow{*} e_2} v_1\ e_1 \underbrace{\xrightarrow[\text{(APP-RIGHT)}]{*}}_{\text{auf } e_2 \xrightarrow{*} v_2} v_1\ v_2 \underbrace{\xrightarrow{*}}_{\text{nach I.A.}} v$
- (BETA-V) Es gelte $(\lambda id.e) \Downarrow v'$ wegen $e[v/id] \Downarrow v'$.
  Nach Induktionsannahme gilt $e[v/id] \xrightarrow{*} v'$.
  Also folgt $(\lambda id.e)v \xrightarrow[\text{(BETA-V)}]{*} e[v/id] \underbrace{\xrightarrow{*}}_{\text{nach I.A.}} v'$.
- (COND-TRUE), (COND-FALSE) analog zu (APP)
- (UNFOLD) analog zu (BETA-V)

„$\Rightarrow$“: Es gelte $e \xrightarrow{*} v$. Zu zeigen: $e \Downarrow v$.

IDEE: $e \xrightarrow{*} v$ geschickt in kürzere small step Folgen aufteilen.
$\rightsquigarrow$ Nach Induktionsannahme existieren big steps für die kürzeren Folgen.
$\rightsquigarrow$ Diese können dann mit den big step Regeln zusammengesetzt werden.

BEWEISTECHNIK: Induktion über die Länge der Berechnungsfolge

Induktionsanfang: $e \xrightarrow{*} v$ in 0 Schritten, d. h. $e = v$ (syntaktisch)
Also gilt $e \Downarrow v$ wegen der big step Regel (VAL).

Induktionsschritt: $e \xrightarrow{+} v$ Fallunterscheidung nach Form von $e$.

1. Fall: $e = \mathbf{rec}\ id.e_1$
   $\rightsquigarrow$ Die Folge kann nur mit (UNFOLD) beginnen, d. h. es gilt
   $e = \mathbf{rec}\ id.e_1 \to e_1[\mathbf{rec}\ id.e/id] \xrightarrow{*} v$
   Da $e_q[...] \xrightarrow{*} v$ kürzer ist, ist die Induktionsannahme anwendbar, d. h.
   $e_1[rec\ id.e_1/id] \Downarrow v$.
   Nach der big step Regel (UNFOLD) folgt $\mathbf{rec}\ id.e_1 \Downarrow v$.

2. Fall: $e = e_1\, e_2$

Dann ist die Folge $e \xrightarrow{*} v$ von der Form $e_1\, e_2 \xrightarrow[\text{(App-Left)}]{*} v_1\, e_2 \xrightarrow[\text{(App-Right)}]{*} v_1\, v_2 \xrightarrow{*} v$

(wobei jede der 3 Teilfolgen leer sein kann).

„⇒“ Applikation:
$e_1\, e_2 \longrightarrow v$ lässt sich aufspalten in $e_1\, e_2 \xrightarrow[\text{(App-Left)}]{*} v_1\, e_2 \xrightarrow[\text{(App-Right)}]{*} v_1\, v_2 \longrightarrow v$

Für $v_1\, v_2 \longrightarrow v$ gibt es folgende Möglichkeiten:

a) Sie kann Länge 0 haben, nämlich im Fall $v_1 = op$. Dann ist $v_1\, v_2 = op\, v_2 \in \mathit{Val}$. In diesem Fall gilt $v_1\, v_2 \Downarrow v_1\, v_2 = v$ wegen (Val).

b) Sie kann mit dem small step (Op) beginnen. Dann ist $v_1 = op\, v'$ und es gilt $op\, v'\, v_2 \to v$ mit (Op).
Also auch $op\, v'\, v_2 \Downarrow v$.

c) Sie kann mit (Beta-V) beginnen, also $v_1\, v_2 = (\lambda id.e')v_2 \to \underbrace{e'[v_w/id] \longrightarrow v}_{\text{um 1 kürzer}}$
⇒ nach Induktionsannahme $e'[v_2/id] \Downarrow v$, also folgt nach der big step Regel (Beta-V) $(\lambda id.e')\, v_2 \Downarrow v$.

Also insgesamt: Es gibt (in allen 4 Fällen) einen big step $v_1\, v_2 \Downarrow v$.
Wenn wir die Induktionsannahme auf die Folgen $e_1 \xrightarrow{*} v_1$ und $e_2 \xrightarrow{*} v_2$ annehmen können, dann folgt $e_1 \Downarrow v_1$ und $e_2 \Downarrow v_2$, also sind alle 3 Prämissen der big step Regel (App) erfüllt und es folgt $e_1\, e_2 \Downarrow v$.

Problem: Wenn $v_1\, v_2 \longrightarrow v$ in 0 Schritten, dann kann es sein, dass $e_1 \xrightarrow{*} v_1$ oder $e_2 \xrightarrow{*} v_2$ nicht kürzer sind als die ursprüngliche Folge (sondern gleich lang).

Also: Länge der Berechnungsfolge ist nicht das richtige „Maß“ für unsere Induktion.

Beobachtung: Im kritischen Fall (d. h. wenn die Berechnungsfolge gleich lang bleibt) nimmt zumindest die Größe des Ausdrucks ab (denn anstelle von $e = e_1\, e_2$ hat man $e_1$ oder $e_2$).

Lösung: Statt Induktion mit $(\mathbb{N}, <)$ benutzt man Induktion über $(\mathbb{N}^2, <_{lex})$, wobei $<_{lex}$ wie folgt definiert ist:
$(m_1, n_1) <_{lex} (m_2, n_2) \Leftrightarrow m_1 < m_2$ oder $(m_1 = m_2\ \mathit{und}\ n_1 < n_2)$

Der Beweis von Satz 3 ist in Ordnung, wenn wir bezüglich der lexigrographischen Ordnung auf $\mathbb{N}^2$ eine Induktion über $\underbrace{\text{Länge der Berechnung } e \longrightarrow v, \text{Größe von } e}_{\in \mathbb{N}^2}$ machen.

Wieso ist diese Induktion erlaubt? $\Longrightarrow$ später!

weitere Fälle: $e = \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2$

Dann ist $e \xrightarrow{*} v$ von der Form
$\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \underbrace{\longrightarrow}_{\text{(Cond-Eval)}} \textbf{if}\ \mathit{true}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to e_1 \xrightarrow{*} v$
$\text{auf } e_0 \xrightarrow{*} \mathit{true}$

oder
$\textbf{if}\ e_0\ \textbf{then} e_1\ \textbf{else}\ e_2 \underbrace{\longrightarrow}_{\text{(Cond-Eval)}} \textbf{if}\ \mathit{false}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to e_2 \xrightarrow{*} v$
$\text{auf } e_0 \xrightarrow{*} \mathit{false}$

Im ersten Fall folgt nach Induktionsannahme: $e_0 \Downarrow \mathit{true}$ und $e_1 \Downarrow v$ (da $e_0 \xrightarrow{*} \mathit{true}$ und $e_0 \xrightarrow{*} v$ kürzer als $e \xrightarrow{*} v$)
Also folgt mit der big step Regel (Cond-True) $e \Downarrow v$. Analog im 2. Fall. □

## 7.2 Noethersche Induktion

IDEE: Wir wollen die Induktion auf $(\mathbb{N}, <)$ verallgemeinern auf sogenannte Noetersche Ordnungen (engl.: *well founded sets*).

ZUR ERINNERUNG: Induktion auf $(\mathbb{N}, <)$
Um zu zeigen, dass $P(n)$ für alle $n \in \mathbb{N}$ gilt, genügt es zu beweisen:

1. Induktionsanfang: $P(0)$
2. Induktionsschritt: Wenn $P(m)$ Für alle $m < n$ gilt, dann gilt auch $P(n)$.

BEMERKUNG: Da es kein $m \in \mathbb{N}$ mit $m < 0$ gibt, ist (1) ein Spezialfall von (2), d. h. es genügt (2).

BEOBACHTUNG: Analoges Induktionsprinzip für $(\mathbb{Z}, <)$ gilt nicht.
Das liegt daran, dass es in $\mathbb{Z}$ unendlich absteigende Folgen $a_0 > a_1 > a_2 ...$ gibt.

ALSO: Nur Bereiche betrachten, in denen es keine unendlich absteigenden Folgen gibt.

**Definition 7:** Sei $A$ eine Menge, $<$ eine zweistellige Relation auf $A$ mit folgenden Eigenschaften:

1. es existiert kein $a \in A$ mit $a < a$ (Irreflexibilität)
2. wenn $a < b$ und $b < c$, dann $a < c$ (Transitivität)

$(A, <)$ heißt Noetersche Ordnung (oder: WFS), wenn es keine unendlich absteigenden Folgen $a_0 > a_1 > a_2 ...$ in $A$ gibt.

**Definition 8:** Sei $<$ eine zweistellige irreflexive und transitive Relation auf der Menge $A$.
Ein Element $a \in A$ heißt minimal (bzgl. $<$), wenn es kein $b \in A$ mit $b < a$ gibt.

**Satz 4:** *Sei $(A, <)$ eine Noethersche Ordnung. Um zu zeigen, dass $P(a)$ für alle $a \in A$ gilt, genügt es zu beweisen:*

1. *Induktionsanfang: $P(a)$ gilt für jedes minimale Element $a \in A$*
2. *Induktionsschritt: Wenn $P(b)$ für alle $b < a$ gilt, dann gilt auch $P(a)$*

BEMERKUNG: Auch hier ist (1) ein Spezialfall von (2), da es für minimale Elemente $a \in A$ keine kleineren gibt.

**Beweis:** Sei (2) für die Eigenschaft $P$ erfüllt.
Angenommen, es existiert ein $a_0 \in A$, für das $P$ nicht gilt.
Dann muss wegen (2) ein $a_1 \in A$ existieren mit $a_1 < a_0$, für das $P$ auch nicht gilt.
Für $a_1$ existiert wieder ein $a_2 \in A$, $a_2 < a_1$, für das $P$ nicht gilt, usw.
$\rightsquigarrow$[2] Es existiert eine unendlich absteigende Folge $a_0 > a_1 > a_2 ...$ in $A$ in Widerstpruch zur Vorraussetzung. □

**Beispiel 18:**

1. $(\mathbb{N}, <)$ ist Noethersch.
2. $(\mathbb{Z}, <)$ ist nicht Noethersch.
3. $(\mathbb{N}^2, <_{lex})$ ist Noethersch.

## 7.3 Wozu big step Semantik?

1. Sie eignet sich nicht zum Beweis der Typsicherheit, da sie nicht zwischen Divergenz und Steckenbleiben unterscheidet.
2. Geeignet für Korrektheitsbeweise von Programmen (da übersichtlicher als small steps).
3. Geeignet als Vorlage für einen Interpreter.

[2] Hier geht Induktion über $\mathbb{N}$ ein, um die Elemente $a_i$ zu erhalten.

**Interpreter mit big step Semantik:**

Wir wissen: Das Ergebnis des big steps $e \Downarrow v$ ist eindeutig durch $e$ bestimmt, d. h. wir haben eine partielle Funktion $eval : Exp \to Val \cup EP$.

$$eval(e) = \begin{cases} r, & \text{falls } e \Downarrow r \\ \text{undef.}, & \text{falls kein solches r existiert} \end{cases}$$

$eval()$ lässt sich rekursiv definieren wie folgt:

$$\left.\begin{array}{l} eval(c) = c \\ eval(op\ n) = op\ n \\ eval(\boldsymbol{\lambda} id.e) = \boldsymbol{\lambda} id.e \end{array}\right\} \quad \text{wegen (VAL)}$$

$eval(op\ n_1\ n_2) = op^I(n_1, n_2)$ wegen (OP)

$eval((\boldsymbol{\lambda} id.e)(v)) = eval(e[v/id])$ wegen (BETA-V)

$$eval(e_1\ e_2) = \begin{cases} eval(v_1\ v_2), & \text{falls } v_1 = eval(e_1), v_2 = eval(e_2) \\ ep, & \text{falls } eval(e_1) = ep \\ & \text{oder } eval(e_1) = e, eval(e_2) = ep \end{cases} \quad \text{wegen (APP)}$$

usw.

$eval()$ kann leicht (in *SML*, *O'Caml*) als funktionales Programm implementiert werden, wobei (OP) und (BETA-V) Vorrang vor (APP) haben müssen, damit man nicht durch (APP) im Falle $e_1 \in Val$ und $e_2 \in Val$ in eine Endlosrekursion geht.

# 8 Umgebungssemantik

BEOBACHTUNG: Dieser Interpreter ist noch sehr ineffizient, da die Substitution in (BETA-V) und (UNFOLD) lineare Laufzeit in der Größe von $e$ hat.

DESHALB: Substitution zur Laufzeit eines Programms sollte vermieden werden.

IDEE: Statt $v$ tatsächlich für jedes Vorkommen von *id* einzusetzen, merkt man sich in einer so genannten Umgebung, dass der Name *id* für $v$ steht. In dieser Umgebung arbeitet man den Ausdruck $e$ (mit frei vorkommenden Namen) weiter ab. Immer wenn man auf *id* stößt, schlägt man in der Umgebung nach und findet dort den Wert $v$. Wenn es für ein- und denselben Namen *id* mehrere Einträge gibt (z. B. durch mehrere Deklarationen für *id*) dann gilt der („zeitlich") letzte Eintrag.

**Beispiel 19:** $(\boldsymbol{\lambda} x.\boldsymbol{\lambda} y.x * x + y * y)\, 2\, 3$
Bei der ersten Parameterübergabe müsste laut (BETA-V) der Wert 2 für x eingesetzt werden, was zum Ausdruck $(\boldsymbol{\lambda} y.2 * 2 + y * y)\, 3$ führt. Stattdessen schreibt man den Eintrag $[x : 2]$ in die aktuelle Umgebung.
Bei der zweiten Parameterübergabe müsste 3 für $y$ eingesetzt werden, stattdessen wird der Eintrag $[y : 3]$ hinzugefügt $\rightsquigarrow$ Umgebung: $[y : 3, x : 2]$

Leider geht es nicht immer so einfach, wie in diesem Beispiel, denn der Wert $v$ in der (BETA-V)-Regel kann selbst schon durch eine Substitution entstanden sein.

**Beispiel 20:** **let** $x = 1$ **in let** $f = \boldsymbol{\lambda} y.y + x$ **in let** $x = 2$ **in** $f\, x$
In subst. Semantik passiert folgendes:

1. 1 wird für x eingesetzt $\rightsquigarrow$ **let** $f = \boldsymbol{\lambda} y.y + 1 \ldots$
2. $\boldsymbol{\lambda} y.y + 1$ wird für $f$ eingesetzt.
3. 2 wird für $x$ eingesetzt $\rightsquigarrow (\boldsymbol{\lambda} y.y + 1)\, 2$
4. Schließlich 2 für $y \rightsquigarrow 2 + 1 \Downarrow 3$

Wie wird dies in Umgebungssemantik realisiert?

1. Zuerst trägt man 1 für $x$ ein $\rightsquigarrow [x : 1]$
2. Dann trägt man $\boldsymbol{\lambda} y.y + x$ zusammen mit der aktuellen Umgebung $[x : 1]$ für $f$ ein $[f : (\boldsymbol{\lambda} y.y + x, [x : 1])^3, x : 1]$
3. Dann trägt man 2 für $x$ ein. $\rightsquigarrow [x : 2, f : (\boldsymbol{\lambda} y.y + x, [x : 1]), x : 1]$
4. In dieser Umgebung wird $f\, x$ ausgewertet, d. h. Wert $x = 2$ wird als Parameter an die Funktion $f = \underbrace{(\boldsymbol{\lambda} y.\ y + x, [x : 1])}_{\text{Darst. von } \boldsymbol{\lambda} y.y+1}$ übergeben.
   Dazu wird $y : 2$ in die alte Umgebung (die aus der closure) eingetragen $\rightsquigarrow [y : 2, x : 1]$
   In dieser Umgebung wird der Rumpf $y + x$ der Funktion $f$ ausgewertet $\rightsquigarrow$ Ergebnins 3.

BEACHTE: Für den "globalen Namen" $x$ im Rumpf von $f$ zählt die Umgebung $[x : 1]$, die zum Deklarationszeitpunkt der Funktion $f$ galt. Globale Namen einer Funktion sind also bei jedem Aufruf der Funktion/Prozedur an die gleichen Werte gebunden. Man spricht deshalb auch von statischer Bindung (engl.: *static scope*). Statische Bindung ist das übliche in blockstrukturierten (imperativen oder funktionalen) Sprachen wie *Algol*, *Pascal*, *ML*, *Scheme*.
Der Gegensatz zur statischen Bindung ist dynamische Bindung: Für die globalen Namen einer Funktion oder Prozedur zählt die Umgebung die zum Aufrufzeitpunkt gilt, d. h. diese Namen

[3] sog. Funktionsabschluss (engl.: *closure*), kann als Darstellung von $\boldsymbol{\lambda} y.y + 1$ aufgefasst werden.

können bei jedem Aufruf an neue Werte gebunden sein. Dynamische Bindung ist die Ausnahme in Programmiersprachen. Sie galt in alten *LISP*-Dialekten und in "Spezialsprachen" wie *emacs-lisp*.
Dynamische Bindung ist schlecht mit Typsystemen vereinbar (denn der globale Name, der seinen Wert ändern kann, kann ja auch seinen Typ verändern) und auch für den Programmierer undurchsichtig (ein- und dieselbe Funktion kann bei verschiedenen Aufrufen unterschiedliche Bedeutungen haben).
In objektorientierten Sprachen hat man das sog. late binding für Methodennamen, das an dynamische Bindung erinnert. Das late binding hat aber nicht mit der Blockstruktur sondern mit Vererbung zu tun (sog. overriding von Methodennamen).

## 8.1 Umgebungssemantik für $\mathcal{L}_2$ (und $\mathcal{L}_0$,$\mathcal{L}_1$)

Die Menge *Env* aller (Laufzeit-)Umgebungen $\eta$, die Menge *Cl* aller closures *cl* und der Definitionsbereich $dom(\eta)$ einer Umgebung $\eta$ sind definiert durch

$$\begin{array}{lcll} \eta & ::= & [id_0 : cl_0, \ldots, id_n : cl_n] \quad (n \geq -1) & \text{mit } dom(\eta) = \{id_0, \ldots, id_n\} \\ cl & ::= & (e, \eta) & \text{mit } free(e) \subseteq dom(\eta) \end{array}$$

**Beispiel 21:**

- $[\,] \in Env$
- $(1, [\,]) \in Cl, (\boldsymbol{\lambda} x.x, [\,]) \in Cl$
- $[f : (\boldsymbol{\lambda} x.x, [\,]), x : (1, [\,])] \in Env$

**Schreibweisen:**

1. Für $\eta = [id_0 : cl_0, \ldots, id_n : cl_n] \in Env$ und $id \in dom(\eta)$ sei
   $\eta(id) = cl_i \qquad \text{mit } i = \min\{j \in \{0, \ldots, n\} \mid id = id_j\}$
2. Für $\eta = [id_0 : cl_0, \ldots, id_n : cl_n] \in Env$, $id \in Id$ und $cl \in Cl$ sei
   $id : cl; \eta = [id : cl, id_0 : cl_0, \ldots, id_n : cl_n]$
3. Wenn $e$ abgeschlossen ist, dann schreibt man $id : e$ statt $id : (e, \eta)$: $[f : \boldsymbol{\lambda} x.x, x : 1]$

Bemerkung: (1) bedeutet, dass wir jede Umgebung als Funktion auffassen können. Dabei können verschiedene Umgebungen für die gleiche Funktion stehen (z. B. zwei Umgebungen, die sich nur in der Reihenfolge der Einträge unterscheiden).

**Definition 9:** Ein big step der Umgebungssemantik ist von der Form $(e, \eta) \Downarrow (v, \eta')$ oder $(e, \eta) \Downarrow (ep, \eta')$, wobei $(e, \eta)$ und $(v, \eta')$ closures sind. Ein big step heißt gültig, wenn er sich mit den folgenden Regeln (und den zugehörigen exception Regeln[4]) herleiten lässt:

(Val) $$(v, \eta) \Downarrow (v, \eta)$$

(Id) $$\frac{\eta(id) \Downarrow cl}{(id, \eta) \Downarrow cl}$$

(Op-1) $$\frac{(e_1, \eta) \Downarrow (op, \eta_1) \quad (e_2, \eta) \Downarrow (n, \eta_2)}{(e_1\, e_2, \eta) \Downarrow (op\, n, [\,])}$$

(Op-2) $$\frac{(e_1, \eta) \Downarrow (op\, n_1, \eta_1) \quad (e_2, \eta) \Downarrow (n_2, \eta_2)}{(e_1\, e_2, \eta) \Downarrow (op^{\mathcal{I}}(n_1, n_2), [\,])}$$

[4] Die exception Regeln werden analog zur big step Substitutionssemantik gebildet.

(Beta-V) $$\frac{(e_1,\eta) \Downarrow (\boldsymbol{\lambda} id.e, \eta_1) \quad (e_2,\eta) \Downarrow cl \quad (e, id:cl;\eta_1) \Downarrow cl'}{(e_1\, e_2, \eta) \Downarrow cl'}$$

(Cond-True) $$\frac{(e_0,\eta) \Downarrow (\mathit{true}, \eta_0) \quad (e_1,\eta) \Downarrow cl}{(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \eta) \Downarrow cl}$$

(Cond-False) $$\frac{(e_0,\eta) \Downarrow (\mathit{false}, \eta_0) \quad (e_2,\eta) \Downarrow cl}{(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \eta) \Downarrow cl}$$

(Unfold) $$\frac{(e, id: (\textbf{rec } id.e, \eta); \eta) \Downarrow cl}{(\textbf{rec } id.e, \eta) \Downarrow cl}$$

Die abgeleitete Regel für **let**-Ausdrücke lautet:

(Let) $$\frac{(e_0,\eta) \Downarrow cl \quad (e_1, id:cl;\eta) \Downarrow cl'}{(\textbf{let } id = e_0 \textbf{ in } e_1, \eta) \Downarrow cl'}$$

**Beispiel 22:** *(für Umgebungssemantik)*

(1) $(\textbf{let } f = \textbf{let } x = 1 \textbf{ in } \boldsymbol{\lambda} y.y + x \textbf{ in } f\,2, (\eta_0)) \Downarrow (3, (\eta_4))$
- (Let)
  $\eta_0 = [\,]$
- (2) $(\textbf{let } x = 1 \textbf{ in } \boldsymbol{\lambda} y.y + x, (\eta_0)) \Downarrow (\boldsymbol{\lambda} y.y + x, (\eta_1))$
  - (Let)
  - (3) $(1, (\eta_0)) \Downarrow (1, (\eta_0))$
    (Val)
  - (4) $(\boldsymbol{\lambda} y.y + x, (\eta_1)) \Downarrow (\boldsymbol{\lambda} y.y + x, (\eta_1))$
    (Val)
    $\eta_1 = [x : (1, (\eta_0))]$
- (5) $(f\,2, (\eta_2)) \Downarrow (3, (\eta_4))$
  - (Beta-V)
    $\eta_2 = [f : (\boldsymbol{\lambda} y.y + x, (\eta_1)) \quad x : (1, (\eta_0))]$
  - (6) $(f, (\eta_2)) \Downarrow (\boldsymbol{\lambda} y.y + x, (\eta_1))$
    - (Id)
    - (7) $(\boldsymbol{\lambda} y.y + x, (\eta_1)) \Downarrow (\boldsymbol{\lambda} y.y + x, (\eta_1))$
      (Val)
  - (8) $(2, (\eta_2)) \Downarrow (2, (\eta_2))$
    (Val)
  - (9) $(y + x, (\eta_3)) \Downarrow (3, (\eta_4))$
    - (Op-2)
      $\eta_3 = [y : (2, (\eta_2)) \quad x : (1, (\eta_0))]$
      $\eta_4 = [\,]$
    - (10) $(y, (\eta_3)) \Downarrow (2, (\eta_2))$
      - (Id)
      - (11) $(2, (\eta_2)) \Downarrow (2, (\eta_2))$
        (Val)
    - (12) $(x, (\eta_3)) \Downarrow (1, (\eta_0))$
      - (Id)
      - (13) $(1, (\eta_0)) \Downarrow (1, (\eta_0))$
        (Val)

Besonderheit des Beispiels: Der Name $x$ ist in der Funktion $f$ „versteckt“, d. h. beim Aufruf $f\,\boldsymbol{\lambda}$ ist der Name $x$ selbst nicht mehr verfügbar (sein Gültigkeitsbereich ist beendet), aber man kann auf $x$ noch „indirekt“ über die Funktion $f$ zugreifen.
Dieses sog. *information hiding* wird interessanter, wenn man Variablen (= Namen für Speicherplätze) hat. Dann kann man Funktionen definieren, die einen „internen Zustand“ haben. Wenn man Tupel von Funktionen zulässt, die alle auf diesen internen Zustand zugreifen können, so kann man Objekte simulieren.

## 8.2 Verbindung zur Substitutionssemantik

Ziel: Äquivalenz zwischen Substitutions- und Umgebungssemantik beweisen.

Idee: Eine Closure $(e, \eta)$ steht für den abgeschlossenen Ausdruck, den man aus $e$ erhält, wenn man alle Substitutionen „gemäß $\eta$“ durchführt.
Also sollte $(e, \eta)$ in der Umgebungssemantik zum „gleichen Resultat“ führen wie der entsprechende abgeschlossene Ausdruck in der Substitutionssemantik.

**Beispiel 23:** $(\textbf{let } x = 1 \textbf{ in } \boldsymbol{\lambda} y.y + x, \eta_0)$ steht für $\textbf{let } x = 1 \textbf{ in } \boldsymbol{\lambda} y.y + x$
$(\boldsymbol{\lambda} y.y + x, \eta_1)$ steht für $\boldsymbol{\lambda} y.y + 1$

**Definition 10:** *(Übersetzungsfunktion)*
Die Übersetzungsfunktion

$$tr : Cl \to Exp$$

wird durch Induktion über die Größe der closures definiert durch

$$tr(e, [id_0 : cl_0, \ldots, id_n : cl_n]) = e[tr(cl_0)/id_0] \ldots [tr(cl_n)/id_n]$$

**Beispiel 24:** $tr(\boldsymbol{\lambda} y.y + x, [x : (1, [\,])]) = (\boldsymbol{\lambda} y.y + x)[tr(1, [\,])/x] = \boldsymbol{\lambda} y.y + 1$

Beachte: Wenn es für einen Namen *id* mehrere Einträge in $\eta$ gibt, dann wirkt nur der am weitesten links stehende (denn wenn *id* einmal durch einen abgeschlossenen Ausdruck ersetzt ist, kommt *id* anschließend nicht mehr vor, d. h. die weiteren Substitutionen wirken nicht mehr).

### Vermuteter Zusammenhang:

1. $(e, \eta) \Downarrow (v, \eta') \Rightarrow tr(e, \eta) \Downarrow tr(v, \eta')$
2. $tr(e, \eta) \Downarrow v \Rightarrow \exists (v', \eta') \in Cl$ mit $(e, \eta) \Downarrow (v', \eta')$ und $tr(v', \eta') = v$

Um diese Zusammenhänge zu beweisen benötigt man eine andere Charakterisierung der Übersetzungsfunktion $tr()$.

**Lemma 5:** *$tr(e, \eta)$ ist abgeschlossen für jede closure $(e, \eta)$ und es gelten die folgenden Gleichungen*

$$\begin{aligned}
tr(c, \eta) &= c \\
tr(id, \eta) &= tr(\eta(id)) \\
tr(e_1 e_2, \eta) &= (tr(e_1, \eta))\,(tr(e_2, \eta)) \\
tr(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \eta) &= \textbf{if } tr(e_0, \eta) \textbf{ then } tr(e_1, \eta) \textbf{ else } tr(e_2, \eta) \\
tr(\boldsymbol{\lambda} id.e, \eta) &= \boldsymbol{\lambda} id.tr(e, \eta \setminus id) \\
tr(\textbf{rec } id.e, \eta) &= \textbf{rec } id.tr(e, \eta \setminus id)
\end{aligned}$$

*wobei $\eta \setminus id$ die Umgebung ist, die aus $\eta$ entsteht, indem man alle Einträge für den Namen id entfernt.*

**Lemma 6:** $tr(e, id : cl; \eta) = tr(e, \eta \setminus id)[tr(cl)/id]$

**Beweis:** $tr(e, id : cl; \eta) = e[tr(cl)/id][\ldots(\text{Subst. für } \eta)]$
$= e[tr(cl)/id][\ldots(\text{Subst für } \eta \setminus id)]$
$= e[\ldots][tr(cl)/id] = tr(e, \eta \setminus id)[tr(cl)/id]$ □

**Satz 5:** *(Korrektheit der Umgebungssemantik)*
*Wenn $(e, \eta) \Downarrow (v, \eta')$ in der Umgebungssemantik, dann gilt $tr(e, \eta) \Downarrow tr(v, \eta')$ in der Substitutionssemantik.*

**Beweis:** Induktion über Länge der Herleitung von $(e, \eta) \Downarrow (v', \eta')$ und Fallunterscheidung nach der zuletzt angewendeten Regel.

(Val) Wenn $(v, \eta) \Downarrow (v, \eta)$ wegen (*Val*), dann gilt auch $tr(v, \eta) \Downarrow tr(v, \eta)$ nach Regel (*Val*) der Substitutionssemantik, weil $tr(v, \eta)$ (nach dem Lemma).

(Id) Es gelte $(id, \eta) \Downarrow cl$ wegen Regel (Id) mit Prämisse $\eta(id) \Downarrow cl$.
Nach Induktionsannahme gilt: $tr(\eta(id)) \Downarrow tr(cl)$ in der Substitutionssemantik.
$tr(\eta(id)) = tr(id, \eta)$ (Lemma), also $tr(id, \eta) \Downarrow tr(cl)$

(Op-1) Es gelte $(e_1\, e_2, \eta) \Downarrow (op\ n_1, [\,])$ wegen Regel (Op-1) mit Prämissen
$(e_1, \eta) \Downarrow (op, \eta_1)$
$(e_2, \eta) \Downarrow (n, \eta_2)$
Dann gilt nach Induktionsannahme:

1. $tr(e_1, \eta) \Downarrow tr(op, \eta_1) = op$
2. $tr(e_2, \eta) \Downarrow tr(n, \eta_2) = n$
   Außerdem gilt wegen (Val) der Substitutionssemantik
3. $op\ n \Downarrow op\ n$

Aus (1),(2) und (3) folgt mit der Regel (App) der Substitutionssemantik $(tr(e_1, \eta))(tr(e_2, \eta)) \Downarrow op\ n$

(Op-2) analog

(Beta-V) Es gelte $(e_1\, e_2, \eta) \Downarrow cl'$ wegen (Beta-V) mit Prämissen
$(e_1, \eta) \Downarrow (\boldsymbol{\lambda} id.e, \eta_1)$
$(e_2, \eta) \Downarrow cl$
$(e, id : cl; \eta_1) \Downarrow cl'$
Nach Induktionsannahme gilt:

1. $tr(e_1, \eta) \Downarrow tr(\boldsymbol{\lambda} id.e, \eta_1) = \boldsymbol{\lambda} id.tr(e, \eta_1 \setminus id)$
2. $tr(e_2, \eta) \Downarrow tr(cl)$
3. $tr(e, id : cl; \eta_1) \Downarrow tr(cl')$
   Daraus ergibt sich mit (Beta-V) der Substitutionssemantik
4. $(\boldsymbol{\lambda} id\, tr(e, \eta_1 \setminus id)).(tr(cl)) \Downarrow tr(cl')$

Aus (1),(2) und (4) ergibt sich mit der Regel (App):
$(tr(e, \eta))\,(tr(e_2, \eta)) \Downarrow tr(cl')$

(Cond-True) trivial.

(Cond-False) trivial.

(Unfold) ähnlich wie (Beta-V) □

**Satz 6:** *(Vollständigkeit der Umgebungssemantik)*
*Wenn* $tr(e,\eta) \Downarrow v$ *in der Substitutionssemantik, dann existiert eine closure* $(v',\eta')$ *mit*

1. $(e,\eta) \Downarrow (v',\eta')$ *in der Umgebungssemantik*
2. $tr(v',\eta') = v$

Beweis: ohne Beweis.

Bemerkung: Es genügt (1) zu beweisen, denn nach Satz 5 folgt daraus: $tr(e,\eta) \Downarrow tr(v',\eta')$. Da die big step Substitutionssemantik (im Endergebnis) deterministisch ist, folgt schließlich $tr(v',\eta') = v$.

**Korollar 1:** *(Äquivalenz der Semantiken für abgeschlossenen Ausdruck)*
*Für einen* abgeschlossenen *Ausdruck e gilt* $e \Downarrow v$ *genau dann, wenn eine closure* $(v',\eta')$ *existiert mit*

1. $(e,[\,]) \Downarrow (v',\eta')$
2. $tr(v',\eta') = v$

In Worten: Um das Ergebnis eines big steps für $e$ zu erhalten startet man $e$ mit der leeren Umgebung und übersetzt das Resultat $(v',\eta')$ mit der Funktion $tr()$, wobei nur im Falle einer $\boldsymbol{\lambda}$-Abstraktion eine „echte“ Übersetzung stattfindet.

Mit Hilfe der Umgebungssemantik lässt sich ein halbwegs effizienter Interpreter implementieren: Man schreibt eine rekursive Funktion $eval : cl \hookrightarrow cl$ die an die Regeln der Umgebungssemantik angelehnt ist.

Problem: In Regel (Id) hat man noch eine Ineffizienz: $eval(id,\eta) = eval(\eta(id)) \leftarrow$ Umgebung $\eta$ muss durchsucht werden bis man den ersten Eintrag für $id$ findet.

Idee: Man kann zur Compilezeit vorausberechnen, an welcher Stelle der Umgebung der Name $id$ zur Laufzeit zu finden ist.

**Beispiel 25:** $(\boldsymbol{\lambda}x.(\boldsymbol{\lambda}y.\underbrace{y}_{0} + \underbrace{x}_{1})\underbrace{x}_{0})\,2,[\,] \rightsquigarrow [x:2] \rightsquigarrow [y:2,x:2]$

Zur Compilezeit kann man den sogenannten de Bruijn-Index für jedes gebundene Vorkommen eines Namens berechnen. Er gibt an, wie viele $\boldsymbol{\lambda}$s (bzw. **let**s und **rec**s) man überspringen muss, um zum zugehörigen bindenden $\boldsymbol{\lambda}$(**let**, **rec**) zu gelangen.
Daraus lässt sich ableiten, an welcher Stelle der Umgebung dieser Name zur Laufzeit steht $\Rightarrow$ Namen sind überflüssig, sowohl in den Ausdrücken als auch in den Umgebungen. $\rightsquigarrow$ Namenlose $\boldsymbol{\lambda}$-Abstraktion (= de Bruijn'sche $\boldsymbol{\lambda}$-Notation)

Formal: Die Menge $dbExp$ aller de Bruijn'schen Ausdrücke $e$ ist definiert durch:

$$e \;::=\; c \mid ui \mid \boldsymbol{\lambda}\_.e \mid e_1\,e_2 \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \mid \textbf{rec } \_.e_1 \mid \textbf{let } \_ = e_0 \textbf{ in } e_1$$

Sei $Id^*$ die Menge aller endlichen Listen $\Gamma : [id_0,\ldots,id_n]$ von Namen. Dann ist die de Bruijn-Übersetzung definiert als eine Funktion: $dbtr : Exp \times Id^* \to dbExp$

$$\begin{aligned}
dbtr(c,\Gamma) &= c \\
dbtr(id,\Gamma) &= \underline{i} \quad \text{falls } i = \Gamma(id) \\
dbtr(e_1\,e_2,\Gamma) &= (dbtr(e_1,\Gamma))\,(dbtr(e_2,\Gamma)) \\
dbtr(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2,\Gamma) &= \textbf{if } dbtr(e_0,\Gamma) \textbf{ then } dbtr(e_1,\Gamma) \textbf{ else } dbtr(e_2,\Gamma) \\
dbtr(\boldsymbol{\lambda}id.e,\Gamma) &= \boldsymbol{\lambda}.dbtr(e,id;\Gamma) \text{analog für } \textbf{let} \text{und } \textbf{rec}
\end{aligned}$$

# 9 Ein einfaches Typsystem für $\mathcal{L}_2$

BISHER: Ungetypte Sprachen $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$, d. h. man kann Ausdrücke bilden, die „vom Typ her sinnlos“ sind, z. B.:

| | | |
|---|---|---|
| 1. $+$ *true false* | (Addition boolescher Werte) | bleiben stecken in der small step Semantik („grobe Programmierfehler“) |
| 2. 1 2 | (Zahl als linke Seite einer App.) | |
| 3. **if** 1 **then** 2 **else** 3 | (Zahl als Bedingung im **if**) | |
| 4. $+$ 1 2 3 4 | ($+$ auf mehr als zwei Argumente) | |

In dynamisch getypten Sprachen (*Lisp*, *Scheme*) sind solche Ausdrücke im Prinzip erlaubt. Sie führen zu Laufzeit(typ)fehlern.
In statisch getypten Sprachen (*Pascal*, *Java*, *ML*) werden sie zur Compile-Zeit abgelehnt, da sie nicht wohlgetypt sind.

## 9.1 Was sind Typen, wozu dienen sie?

- zur besseren Organisation von Programmen
- zum Entwurf eines effizienten Compilers (der Compiler kann oft am Typ einer Variablen erkennen, wieviel Speicherplatz allokiert werden muss)
- um grobe „Programmierfehler“ abzufangen

AUS THEORETISCHER SICHT: Programme, die stecken bleiben, sollen von vornherein ausgeschlossen werden (zur Compilezeit). Diese sog. Typsicherheit lässt sich beweisen.

AUS PRAKTISCHER SICHT: Typen helfen dem Programmierer, grobe Fehler zu vermeiden, z. B. eine Funktion mit Argumenten vom falschen Typ, in falscher Reihenfolge oder falscher Anzahl aufzurufen.
In statisch getypten Sprachen werden solche groben Fehler zur Compile-Zeit entdeckt.

**Definition 11:** Die Menge *Type* aller Typen $\tau$ ist definiert durch:

$$\begin{array}{rcll} \tau & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{unit} & \text{Basistypen} \\ & \mid & \tau_1 \to \tau_2 & \text{Funktionstypen} \end{array}$$

INTUITION: Ein Element vom Typ $\tau_1 \to \tau_2$ ist eine Funktion, die ein Argument von Typ $\tau_1$ besitzt und ein Resultat vom Typ $\tau_2$ liefert.

KONKRETE SYNTAX: „$\to$“ ist rechtsassiziativ, d. h. $\tau_1 \to \tau_2 \to \tau_3$ steht für $\tau_1 \to (\tau_2 \to \tau_3)$.

## 9.2 Typen von Konstanten

Man schreibt „$c :: \tau$“, um festzulegen, dass die Konstante $c$ den Typ $\tau$ hat.

(UNIT) $()$ :: $\mathit{unit}$
(BOOL) $b$ :: $\mathit{bool}$
(INT) $n$ :: $\mathit{int}$
(AOP) $op$[5] :: $\mathit{int} \to \mathit{int} \to \mathit{int}$
(ROP) $op$[6] :: $\mathit{int} \to \mathit{int} \to \mathit{bool}$

[5] aritmetische Operatoren
[6] relationale Operatoren

## 9.3 Typen von Ausdrücken

PROBLEM: Um zu entscheiden, ob ein Ausdruck $x + 1$ wohlgetypt ist, muss man wissen, welchen Typ der Name $x$ hat. Im Allgemeinen bedeutet das, dass man die Typen der frei vorkommenden Namen kennen muss.
$\rightsquigarrow$ Man muss Typinformation über alle Namen sammeln, die vorher deklariert oder als Parameter eingeführt wurden.
$\rightsquigarrow$ „Tabelle“, in der man sich die Typen von Namen merkt.

## 9.4 Typherleitungen

**Definition 12:** Eine Typumgebung ist eine endliche partielle Funktion $\Gamma : Id \hookrightarrow Type$ (d. h. eine Funktion mit endlichem Definitionsbereich $dom(\Gamma)$

DARSTELLUNG: $\Gamma$ kann als Liste dargestellt werden: $[id_1 : \tau_1, \ldots, id_n : \tau_1]$ ist die Funktion $\Gamma$ mit $dom(\Gamma) = \{id_1, \ldots, id_n\}$ und $\Gamma(id_i) = \tau_i$ für $i = 1, \ldots, n$.

**Definition 13:** Ein Typurteil ist eine „Formel“ der Gestalt $\Gamma \rhd e :: \tau$ (lies: „e hat Typ $\tau$ in der Typumgebung $\Gamma$“).

Wir benötigen einen Mechanismus, mit dem wir „gültige“ Typurteile herleiten können $\rightsquigarrow$ sog. Typregeln.

SCHREIBWEISE: Sei $f : A \hookrightarrow B$ eine partielle Funktion, sei $a \in A$, $b \in B$.
Dann bezeichnet $f[b/a]$ die Funktion $g : A \hookrightarrow B$ mit:
$dom(g) = dom(f) \cup \{a\}$

$$g(a') = \begin{cases} b, & \text{falls } a' = a \\ f(a'), & \text{sonst} \end{cases}$$

In Worten: $f[b/a]$ ist die Funktion, die sich von $f$ darin unterscheidet, dass sie an der Stelle $a$ den Wert $b$ annimmt.

**Definition 14:** Ein Typurteil heißt gültig, wenn es mit den folgenden Regeln hergeleitet werden kann:

(CONST) $$\frac{c :: \tau}{\Gamma \rhd c :: \tau}$$
Jede Konstante hat (in jeder Umgebung) ihren vordefinierten Typ.

(ID) $\Gamma \rhd id :: \tau$ falls $id \in dom(\Gamma)$ und $\Gamma(id) = \tau$
Ein Name hat den in $\Gamma$ eingetragenen Typ.

(APP) $$\frac{\Gamma \rhd e_1 :: \tau \to \tau' \quad \Gamma \rhd e_2 :: \tau}{\Gamma \rhd e_1 e_2 :: \tau'}$$
Wenn $e_1$ den Typ $\tau \to \tau'$ und $e_2$ den dazu passenden Typ $\tau$ hat, dann liefert die Applikation $e_1\, e_2$ ein Resultat vom Typ $\tau'$.

(COND) $$\frac{\Gamma \rhd e_0 :: \mathit{bool} \quad \Gamma \rhd e_1 :: \tau \quad \Gamma \rhd e_2 :: \tau}{\Gamma \rhd \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \;\; :: \tau}$$

(ABSTR) $$\frac{\Gamma[\tau/id] \rhd e :: \tau'}{\Gamma \rhd \boldsymbol{\lambda} id.e :: \tau \to \tau'}$$
Wenn $e$ den Typ $\tau'$ hat unter der (zusätzlichen) Annahme, dass der Parameter $id$ vom Typ $\tau$ ist, dann hat die $\boldsymbol{\lambda}$-Abstraktion den Typ $\tau \to \tau'$.

(REC) $$\frac{\Gamma[\tau/id] \rhd e :: \tau}{\Gamma \rhd \textbf{rec } id.e \;\; :: \tau}$$
Wenn $e$ den Typ $\tau$[7] hat unter der Annahme, dass $id$ vom Typ $\tau$ ist, dann ist auch **rec** $id.e$ vom Typ $\tau$.

[7] Da wir meistens nur Funktionen rekursiv definieren, ist $\tau$ üblicherweise ein Funktionstyp.

**Abgeleitete Typregel für let-Ausdrücke:**

$$\text{(Let)} \qquad \frac{\Gamma \rhd e_1 :: \tau_1 \quad \Gamma[\tau_1/id] \rhd e_2 :: \tau_2}{\Gamma \rhd \textbf{let } id = e_1 \textbf{ in } e_2 :: \tau_2}$$

Herleitung: $\textbf{let } id = e_1 \textbf{ in } e_2$ steht für $(\boldsymbol{\lambda} id.e_2)e_1$

1. Prämisse: $\Gamma \rhd e_1 :: \tau_1$
2. Prämisse: $\Gamma[\tau_1/id] \rhd e_2 :: \tau_2 \xrightarrow[\text{(Abstr)}]{} \Gamma \rhd \boldsymbol{\lambda} id.e_2 :: \tau_1 \to \tau_2$

Daraus folgt mit (App): $\Gamma \rhd (\boldsymbol{\lambda} id.e_2)e_1 :: \tau_2$, also $\Gamma \rhd \textbf{let } id = e_1 \textbf{ in } e_2 :: \tau_2$

**Beispiel einer Typherleitung (Fakultät):**

siehe Extrablatt *Typherleitung: Fakultätsfunktion.*

## 9.5 Wohlgetyptheit

**Definition 15:** $e$ heißt wohlgetypt in der Typumgebung $\Gamma$, wenn (mindestens) ein gültiger Typ $\tau$ existiert mit $\Gamma \rhd e :: \tau$.

Bemerkung: Ein Ausdruck kann (bzgl. einer Typumgebung $\Gamma$) mehr als einen Typ haben, z. B.

- $[\,] \rhd \boldsymbol{\lambda} x.x :: \mathit{int} \to \mathit{int}$
- $[\,] \rhd \boldsymbol{\lambda} x.x :: \mathit{bool} \to \mathit{bool}$
- $[\,] \rhd \boldsymbol{\lambda} x.x :: \tau \to \tau$ für jeden Typ $\tau$

Beachte: Die Typregeln liefern noch keinen Algorithmus zur Überprüfung der Wohlgetyptheit, sondern nur ein „Beweisverfahren“ für Typurteile (da man bei (Abstr) und (Rec) raten muss)!

Ziel: Wir wollen beweisen, dass die Berechnung eines abgeschlossenen wohlgetypten Ausdrucks niemals steckenbleibt.

**Lemma 7:** *(Zusammenhang zwischen frei vorkommenden Namen und Typurteilen)*

*(a) Wenn $\Gamma \rhd e :: \tau$, dann gilt $\mathit{free}(e) \subseteq \mathit{dom}(\Gamma)$.*

*(b) Wenn $\Gamma$ und $\Gamma'$ auf $\mathit{free}(e)$ definiert sind und übereinstimmen, dann gilt $\Gamma \rhd e :: \tau \Leftrightarrow \Gamma' \rhd e :: \tau$*

*(c) Wenn $\Gamma$ auf $\mathit{free}(e)$ definiert ist und $\Gamma'$ ist die Einschränkung von $\Gamma$ auf $\mathit{free}(e)$, dann gilt: $\Gamma \rhd e :: \tau \Leftrightarrow \Gamma' \rhd e :: \tau$*

Intuition:

(a) bedeutet, dass $e$ nur wohlgetypt sein kann, wenn jeder in $e$ frei vorkommenden Name „bekannt“ ist (d. h. in $\Gamma$ eingetragen)

(b) bedeutet, dass nur die Typen der in $e$ frei vorkommenden Namen bekannt sein müssen (die Typen anderer Namen spielen keine Rolle)

(c) ist ein Spezialfall von (b): Typen von Namen, die nicht frei in $e$ sind, können aus $\Gamma$ entfernt werden.

**Beweis:** Einfache Induktion über die Länge der Herleitung von $\Gamma \rhd e :: \tau$ oder die Größe von $e$ (hier nicht explizit durchgeführt). □

**Lemma 8:** *(Frei vorkommende Namen und Substitution)*
*Es gilt $\mathit{free}(e[e'/id]) \subseteq (\mathit{free}(e) \setminus \{id\}) \cup \mathit{free}(e')$*

BEGRÜNDUNG: Da jedes freie Vorkommen von $id$ durch $e'$ ersetzt wird, muss $id$ zunächst aus den frei vorhandenen Namen entfernt werden. Dann kommen die in $e'$ frei vorkommenden Namen hinzu. Da $id$ nicht unbedingt in $e$ vorkommt, gilt aber nur „$\subseteq$“ statt „$=$“.

SPEZIALFALL: Wenn $e'$ abgeschlossen ist, so gilt $free(e[e'/id]) \subseteq free(e) \setminus \{id\}$

**Lemma 9:** *(Substitution und Typurteile)*
*Wenn $\Gamma \rhd e :: \tau$ und $\Gamma[\tau/id] \rhd e' :: \tau'$, dann gilt $\Gamma \rhd e'[e/id] :: \tau'$.*

IN WORTEN: *Wenn $e'$ den Typ $\tau'$ hat und unter der Annahme, dass id vom Typ $\tau$ ist, dann kann man jeden Ausdruck $e$ vom gleichen Typ $\tau$ für id einsetzen, ohne dass sich der Typ von $e'$ verändert.*

KURZ: Bei einer „vernünftigen“ Substitution bleiben Wohlgetyptheit und Typ eines Ausdrucks erhalten.

BEACHTE: Das Lemma gilt nicht, wenn man *naiv* substituiert.

BEWEISSKIZZE: Man betrachte eine Herleitung für das Typurteil $\Gamma[\tau/id] \rhd e' :: \tau'$.
Während dieser Herleitung stößt man auf die freien Vorkommen von $id$ in $e'$. Bei jedem solchen freien Vorkommen schlägt man in der Typumgebung nach und findet den Typ $\tau$ für $id$ (denn bei einem freien Vorkommen ist der Eintrag $id : \tau$ nicht überschrieben).
Eine Herleitung für $\Gamma \rhd e'[e/id] :: \tau'$ verläuft ganz analog: Überall, wo man vorher auf ein freies Vorkommen von $id$ gestoßen ist, trifft man jetzt auf $e$. Also kann man jetzt (statt $\tau$ für $id$ nachzuschlagen) die Herleitung $\Gamma \rhd e :: \tau$[8] einfügen.

Diese Argumentation ist noch unsauber, da sich während der Herleitung die Typumgebung $\Gamma$ verändern kann, d. h. statt $\Gamma \rhd e :: \tau$ muss man $\Gamma' \rhd e :: \tau$ beweisen, wobei $\Gamma'$ aus $\Gamma$ durch Hinzufügen oder Überschreiben von Einträgen entstanden ist.
Wenn dabei nur Namen eingetragen werden, die nicht frei in $e$ sind, dann ist alles ok, weil nach dem vorhergehenden Lemma auch $\Gamma' \rhd e :: \tau$ gilt.
Wenn ein Name hinzugekommen ist, der frei in $e$ ist, dann kann $\Gamma' \rhd e :: \tau$ schiefgehen. Aber dies wäre eine Namenskollision, die wir in der Definition der Substitution ausschließen (durch gebundene Umbenennung).

**Satz 7:** *(Typerhaltung, „Preservation“)*
*Wenn $\Gamma \rhd e :: \tau$ und $e \to e'$, dann gilt auch $\Gamma \rhd e' :: \tau$ (d. h. die Wohlgetyptheit und der Typ[9] eines Ausdrucks bleiben bei jedem small step erhalten).*

**Beweis:** Es gelte $\Gamma \rhd e :: \tau$ und $e \to e'$.
Man beweist $\Gamma \rhd e' :: \tau$ durch Induktion über die Länge der Herleitung von $\Gamma \rhd :: \tau$ und durch Fallunterscheidung nach der (zuletzt angewandten) small step Regel.

1. Fall: $op\ n_1\ n_2 \to op^{\mathcal{I}}(n_1, n_2)$ mit Regel (OP).
Wenn $op \in \{+, -, *, /, \mathbf{mod}\}$, dann ist $op :: \mathit{int} \to \mathit{int} \to \mathit{int}$, also $\Gamma \rhd op\ n_1\ n_2 :: \mathit{int}$ und per Definition von $op^{\mathcal{I}}$ gilt $op^{\mathcal{I}}(n_1, n_2) \in \mathit{int} \cup EP$. Im Falle $op^{\mathcal{I}}(n_1, n_2) \in \mathit{int}$ ist $\Gamma \rhd op^{\mathcal{I}}(n_1, n_2) :: \mathit{int}$, im anderen Fall ist nichts zu zeigen.
Wenn $op \in =, <, \ldots$, analog mit $\mathit{bool}$.

2. Fall: $(\boldsymbol{\lambda} id.e)\ v \to e[v/id]$ mit (BETA-V).
$\Gamma \rhd (\boldsymbol{\lambda} id.e)\ v :: \tau$, kann nur mit Typregel (APP) folgen aus (1) $\Gamma \rhd \boldsymbol{\lambda} id.e :: \tau' \to \tau$ und (2) $\Gamma \rhd v :: \tau'$. (1) kann nur mit (ABSTR) folgen aus (3) $\Gamma[\tau/id] \rhd e :: \tau$.
Aus (2) und (3) folgt nach dem Lemma: $\Gamma \rhd e[v/id] :: \tau$

[8] stimmt nicht ganz
[9] genauer: jeder mögliche Typ

3. Fall: $e_1\,e_2 \to e_1'\,e_2$ gelte wegen $e_1 \to e_1'$ mit (App-Left).
$\Gamma \rhd e_1\,e_2 :: \tau$, kann nur mit (APP) aus $\Gamma \rhd e_1 :: \tau' \to \tau$ und $\Gamma \rhd e_2 :: \tau'$ entstanden sein.
Da $\Gamma \rhd e_1 :: \tau' \to \tau$ eine kürzere Herleitung hat, ist die Induktionsannahme anwendbar, also folgt $\Gamma \rhd e_1' :: \tau' \to \tau$.
Daraus ergibt sich mit der Typregel (APP) wieder $\Gamma \rhd e_1'\,e_2 :: \tau$.

4. Fall: $v\,e_1 \to v\,e_1'$ mit (App-Right) wegen der Prämisse $e_1 \to e_1'$.
Wenn $\Gamma \rhd v\,e_1 :: \tau$, dann kann dies nur mit (App) aus $\Gamma \rhd v :: \tau' \to \tau$ und $\Gamma \rhd e_1 :: \tau'$ entstanden sein.
Da $\Gamma \rhd e_1 :: \tau'$ eine kürzere Herleitung hat, ist die Induktionsannahme anwendbar, also folgt $\Gamma \rhd e_1' :: \tau'$.
Es folgt wieder mit (App): $\Gamma \rhd ve_1' :: \tau$

5. Fall: **rec** $id.e \to e[$**rec** $id.e/id]$ mit (Unfold)
Wenn $\Gamma \rhd$ **rec** $id.e :: \tau$, dann kann dies nur mit (Rec) aus $\Gamma[\tau/id] \rhd e :: \tau$ entstanden sein. Nach dem Lemma über Typerhaltung bei Substitution folgt $\Gamma \rhd e[$**rec** $id.e/id] :: \tau$.

weitere Fälle: (Cond-True), (Cond-False), (Cond-Eval): Übung. □

Preservation bedeutet, dass zur Laufzeit keine „Typfehler“ entstehen (ein wohlgetypter Ausdruck kann nicht zu einem falsch getypten umgeformt werden).
Noch zu zeigen: Es gibt auch keine andere Art von „steckenbleiben“.

**Lemma 10:** *(wohlgetypte abgeschlossene Werte)*

1. *Wenn $v$ :: **int** (d. h. $v$ abgeschlossen vom Typ **int**), dann ist $v$ eine Zahl $n \in$ **int**.*
2. *Wenn $v$ :: **bool** (d. h. $v$ abgeschlossen vom Typ **bool**), dann ist $v$ ein boolescher Wert $b \in \{true, false\}$.*
3. *Wenn $v :: \tau \to \tau'$ (d. h. $v$ abgeschlossen vom Funktionstyp), dann ist $v$ von der Form $\boldsymbol{\lambda} id.e$ oder $op$ oder $op\ n$.*

**Beweis:** Die einzigen Werte in $\mathcal{L}_2$ sind von der Form

$$\left.\begin{array}{l} b \in Bool \\ () \in Unit \\ n \in Int \\ op \in Op \end{array}\right\} \text{ Konstanten } c$$

$\boldsymbol{\lambda} id.e$
$op\ v_1$

Daraus folgt sofort: Nur $n$ kann Typ **int** haben, nur $b$ kann Typ **bool** haben.
Bleibt (3) zu zeigen: Nur $op$, $\boldsymbol{\lambda} id.e$ und $op\ v$ kommen für Funktionstypen in Frage. □

**Satz 8:** *(Existenz eines small steps, „Progress“)*
*Wenn $e$ wohlgetypt und abgeschlossen ist (d. h. $[] \rhd e :: \tau$), dann ist entweder $e \in Val$ oder es existiert ein small step $e \to e'$ oder $e \to \uparrow exn$.*

**Beweis:** Idee: In jeder Situation greift eine der small step Regeln.

Formal: Induktion über die Länge der Herleitung von $e :: \tau$ (oder Größe von $e$) und Fallunterscheidung nach der Form von $e$.

1. Fall: $e = e_1\,e_2$
$e_1\,e_2$ :: $\tau$ kann nur mit der Regel (App) entstanden sein aus Prämissen der Form $e_1 :: \tau' \to \tau$ und $e_2 :: \tau'$.
Wenn $e_1 \notin Val$, dann ist die Induktionsannahme auf (1) anwendbar, also nach $e_1$ einen small step $e_1 \to e_1'$ oder $e_1 \to \uparrow exn$.

Also folgt mit (App-Left) oder (App-Left-Exn) $e_1\,e_2 \to e_1'\,e_2$ bzw. $e_1\,e_2 \to \uparrow exn$.
Wenn $e_1 = v \in \mathit{Val}$ und $e_2 \notin \mathit{Val}$, dann ist die Induktionsannahme auf (2) anwendbar und man argumentiert analog mit (App-Right).
Wenn $e_1 = v_1 \in \mathit{Val}$ und $e_2 = v_2 \in \mathit{Val}$, dann gibt es (nach dem Lemma über wohlgetypte abgeschlossene Werte) folgende Möglichkeiten für $v_1$ und $v_2$:

- $v_1 = op :: \mathbf{int} \to \mathbf{int} \to \beta$ ($\beta \in \{\mathbf{int}, \mathbf{bool}\}$)
  $v_2 :: \mathbf{int}$, also $v_2 = n \in \mathbf{int}$
  Also $e = v_1\,v_2 = op\ n \in \mathit{Val}$
- $v_1 = op\ n_1 :: \mathbf{int} \to \beta$
  $v_2 = n_2 :: \mathbf{int}$
  Also $e = op\ n_1\,n_2 \to op^{\mathcal{I}}(n_1, n_2)$ wegen (Op)
- $v_1 = \boldsymbol{\lambda} id.e :: \tau' \to \tau$
  $v_2 :: \tau$ Also $e = (\boldsymbol{\lambda} id.e)\,v_2 \to e[v_2/id]$ wegen (Beta-V).

andere Fälle: analog (siehe Übung). □

**Satz 9:** *(Typsicherheit, „Safety“)*
*Wenn $e :: \tau$ (d. h. $e$ ist wohlgetypt und abgeschlossen), dann kann die Berechnung für $e$ nicht steckenbleiben, d. h. sie kann nur*

1. *divergieren*
2. *oder mit einer exception terminieren*
3. *oder mit einem Wert terminieren.*

**Beweis:** Idee: „Safety = Progress + Preservation“
Sei $e = e_0 \to\to \cdots \to e_i \to \cdots$ die Berechnung für $e$.
Wenn sie unendlich ist oder wenn sie mit einer Exception endet, dann liegt (1) oder (2) vor.
Es bleibt zu zeigen: Wenn sie mit einem Ausdruck endet, d. h. $e = e_0 \to e_1 \to \cdots \to e_n \not\to$, dann ist dieser Ausdruck $e_n$ ein Wert.
Mit Preservation gilt zunächst: Da $e_0 :: \tau$, ist auch $e_i :: \tau$ für $i = 1, \ldots, n$ (Induktion über $i$).
Dann folgt mit Progress, dass $e_n \in \mathit{Val}$ (denn andernfalls müsste laut Progress ein weiterer small step existieren). □

Frage: Gibt es falsch getypte Ausdrücke, die „gut“ sind, also nicht steckenbleiben?

**Beispiel 26:**

1. **if** *true* **then** 1 **else** *false* terminiert mit 1, ist aber nicht wohlgetypt (da **then**- und **else**-Teil unterschiedlichen Typ haben).
2. $(\boldsymbol{\lambda} x.x\,x)(\boldsymbol{\lambda} x.x\,x)$ divergiert
3. **let** $f = \boldsymbol{\lambda} x.x$ **in if** $f$ *true* **then** $f\,1$ **else** $f\,2$ terminiert mit 1, ist aber nicht wohlgetypt, denn bei der Deklaration **let** $f = \boldsymbol{\lambda} x.x$ hat man zwar eine Auswahl, z. B. $f : \mathbf{int} \to \mathbf{int}$ oder $f : \mathbf{bool} \to \mathbf{bool}$, aber man muss sich zum Deklarationszeitpunkt von $f$ festlegen (da unser Typsystem keine Polymorphie enthält).
   Der Ausdruck ist z. B. in *O'Caml* und *SML* wohlgetypt.

$\rightsquigarrow$ Unser Typsystem lässt sich noch verbessern.

Frage: Ist es möglich, das Typsystem so weit zu „verbessern“, dass man (zur Compilezeit) genau die steckenbleibenden Ausdrücke ablehnt, und alle übrigen akzeptiert?

Antwort: Nein, denn Steckenbleiben ist (ähnlich wie das Halteproblem) eine nichttriviale, und damit nach dem Satz von Rice eine unentscheidbare semantische Eigenschaft.

FOLGERUNG: Diese Eigenschaft lässt sich nicht zur Compilezeit entscheiden.

BEMERKUNG: Es ist auch nicht unbedingt wünschenswert, das Typsystem so flexibel wie möglich zu machen.

**Beispiel 27:** $(\boldsymbol{\lambda} x.x\,x)(\boldsymbol{\lambda} x.x\,x)$ wäre in einem Typsystem mit (uneingeschränkten) rekursiven Typen wohlgetypt, aber es wird in den meisten Programmiersprachen nicht zugelassen, weil man eher einen Programmierfehler vermutet, wenn eine Funktion auf sich selbst angewendet wird.

Andererseits möchte man Beispiel (3) eher doch zulassen $\rightsquigarrow$ Polymorphie! (später)

BEMERKUNG: Indirekt haben wir durch Typsicherheit noch etwas mehr bewiesen: Wenn $e :: \tau$, dann erkennt man am Typ $\tau$, wie man $e$ verwenden darf (ohne dass das Programm stecken bleibt).

**Beispiel 28:** Wenn $e$ :: ***int*** $\rightarrow$ ***int***, dann kann man $e$ auf jedes Argument von Typ ***int*** anwenden und man erhält daraus ein Resultat vom Typ ***int*** (wenn $e$ nicht divergiert oder eine Exception wirft).

# 10 Algorithmen zur Überprüfung der Wohlgetyptheit

BISHER: Rein mathematische Definition der Wohlgetyptheit: $e$ heißt wohlgetypt in $\Gamma$, wenn sich $\Gamma \rhd e :: \tau$ für einen Typ $\tau$ herleiten lässt.
Diese Definition ist geeignet, um Eigenschaften wohlgetypter Ausdrücke zu beweisen, z. B. Typsicherheit: Die Berechnung eines abgeschlossenen wohlgetypten Ausdrucks bleibt nicht stecken.

PROBLEM: In der Praxis benötigt man einen Algorithmus zur Überprüfung der Wohlgetyptheit (oder besser: zur Bestimmung des Typs bzw. aller möglichen Typen eines Ausdrucks). Einen solchen Algorithmus gibt die mathematische Definition (noch) nicht her, weil man bei den Rückwärtsherleitungen an manchen Stellen den passenden Typ erraten muss: In den Regeln (ABSTR) und (REC) lässt sich der Typ $\tau$, der für $id$ in $\Gamma$ eingetragen wird, nicht aus dem Ausdruck $\boldsymbol{\lambda} id.e$ bzw. **rec** $id.e$ ablesen. Wie lässt sich dieses Raten vermeiden?

## 10.1 Mögliche Lösungen

1. Der Programmierer muss bei Parametern ($\boldsymbol{\lambda}$-Abstraktion) und bei rekursiven Funktionen (**rec**-Ausdrücke) einen Typ angeben. Wenn dies nicht der passende Typ ist, dann wird der Ausdruck abgelehnt.
   $\rightsquigarrow$ einfacher Algorithmus, nämlich (eindeutige) Rückwärtsherleitung von Typurteilen.
   Man spricht hier von Typüberprüfung (engl.: *type checking*), denn im Algorithmus wird nur überprüft, ob die vom Programmierer angegebenen Typen die „richtigen“ sind. Dieses Verfahren wird in den traditionellen imperativen Sprachen wie *Pascal*, *C*, *C++* und *Java* verwendet.

2. Man vermeidet das Raten, indem man für die noch nicht bekannten Typen in (ABSTR) und (REC) zunächst „Platzhalter“ einführt, sog. Typvariablen.
   $\rightsquigarrow$ Bei der Rückwärtsherleitung entstehen Typgleichungen, die Typvariablen enthalten. Diese Gleichungen löst man mit dem sog. Unifikationsalgorithmus (aus Prädikatenlogik bekannt).
   Im schlechtesten Fall ist dieser Algorithmus hochgradig ineffizient, in der Praxis hat er sich trotzdem bewährt (da ein so schlechter Fall meist nicht auftritt).
   Der Algorithmus wird in vielen funktionalen Sprachen (in Verbindung mit Polymorphie) angewendet, insbesondere in *SML*, *O'Caml*, *TPML*. Man spricht hier von Typinferenz, weil der Compiler selbst die Typen für Parameter und rekursive Funktionen herausfinden muss.

(2) lässt sich leicht verallgemeinern: Man kann dem Programmierer erlauben, Typen anzugeben (bei manchen Parametern). Das ist sinnvoll, da Typen auch zum Verständnis des Programms beitragen.

## 10.2 Typüberprüfung

Wir definieren die explizit getypte Sprache $\mathcal{L}_2^t$ wie folgt:
In der KFG ersetzen wir die Produktionen für $\boldsymbol{\lambda}$- und **rec**-Ausdrücke durch:

$$\begin{array}{rcl} e & ::= & \boldsymbol{\lambda} id : \tau.e \\ & | & \textbf{rec}\ id : \tau.e \end{array}$$

$\rightsquigarrow$ Es muss einen Typ hinter $\boldsymbol{\lambda} id$ bzw. **rec** $id$ stehen.

Entsprechend wird der syntaktische Zucker verändert:

- **let** $id\ (id_1 : \tau_1) \ldots (id_n : \tau_n) = e_1$ **in** $e_2$
  steht für
  **let** $id = \boldsymbol{\lambda} id_1 : \tau_1. \ldots \boldsymbol{\lambda} id_n : \tau_n.e_1$ **in** $e_2$

- **let rec** $id : \tau = e_1$ **in** $e_2$
  steht für
  **let** $id =$ **rec** $id : \tau = e_1$ **in** $e_2$
- **let rec** $id\ (id_1 : \tau_1) \ldots (id_n : \tau_n) : \tau' = e_1$ **in** $e_2$
  steht für
  **let rec** $id.\tau_1 \to \cdots \to \tau_n \to \tau' = \boldsymbol{\lambda} id_1 : \tau_1. \ldots\ \boldsymbol{\lambda} id_n : \tau_n.e_1$ **in** $e_2$

**Beispiel 29:**
**let rec** $fact(x : int) : int =$ **if** $x = 0$ **then** $1$ **else** $x * fact\,(x-1)$ **in** $fact\,3$
steht für
**let rec** $fact : int \to int =$ $\boldsymbol{\lambda} x : int$**.if** $x = 0$ **then** $1$ **else** $x * fact\,(x-1)$ **in** $fact\,3$
und das steht wiederum für
**let** $fact =$ **rec** $fact : int \to int.\boldsymbol{\lambda} x : int$**.if** $x = 0$ **then** $1$ **else** $x * fact\,(x-1)$ **in** $fact\,3$

BEACHTE: In der Kernsyntax muss der Funktionstyp hinter dem Funktionsnamen stehen. Beim syntaktischen Zucker gibt man nur den Ergebnistyp an, weil die Argumenttypen schon bei den Parametern stehen.
**let**-Ausdrücke behalten ihre bisherige Form: **let** $id = e_1$ **in** $e_2$ (ohne Typ)
$\Rightarrow$ **let**-Ausdrücke können in $\mathcal{L}_2^t$ nicht als syntaktischer Zucker für Aufrufe von $\boldsymbol{\lambda}$-Abstraktionen aufgefasst werden.

### Typregeln für $\mathcal{L}_2^t$:

Die Regeln (ABSTR) und (REC) werden ersetzt durch:

(T-ABSTR) $$\frac{\Gamma[\tau/id] \rhd e :: \tau'}{\Gamma \rhd \boldsymbol{\lambda} id : \tau.e :: \tau \to \tau'}$$

(T-REC) $$\frac{\Gamma[\tau/id] \rhd e\ :: \tau}{\Gamma \rhd \textbf{rec}\ id : \tau.e\ :: \tau}$$

d. h. der Typ, der in $\Gamma$ eingetragen wird, muss nicht mehr „erraten“ werden, sondern wird vom Programmierer angegeben. Die Regel (LET) sieht wie bisher aus, ist aber jetzt eine eigenständige Typregel (keine abgeleitete Regel).

**Satz 10:** *(Eindeutigkeit des Typs und der Typherleitung in $\mathcal{L}_2^t$)*
*Für jede Typumgebung $\Gamma$ und jeden Ausdruck $e \in \mathcal{L}_2^t$ existiert höchstens ein Typ $\tau \in Type$ mit $\Gamma \rhd e :: \tau$ und darüber hinaus ist die Herleitung für $\Gamma \rhd e :: \tau$ eindeutig.*

BEWEISIDEE: In jeder Situation ist höchstens eine Typregel rückwärts anwendbar und jede solche Rückwärtsanwendung führt zu eindeutigen Prämissen (da man keine Typen mehr erraten muss).

FORMAL: Induktion über die Größe von $e$ oder Länge der existierenden Herleitung $\Gamma \rhd e :: \tau$ (nicht explizit durchgeführt).

### Algorithmus zur Typüberprüfung:

Sei *TEnv* die Menge aller Typumgebungen $\Gamma$ und *Exp* die Menge aller gültigen Ausdrücke in $\mathcal{L}_2^t$. Dann existiert nach Satz 10 eine Funktion

$type : TEnv \times Exp \to Type \cup \{error\}$ mit

$$type(\Gamma, e) = \begin{cases} \tau, & \text{falls } \Gamma \rhd e :: \tau \\ error, & \text{sonst, d. h. wenn kein } \tau \text{ mit } \Gamma \rhd e :: \tau \text{ existiert} \end{cases}$$

Aus den Typregeln lässt sich eine induktive Definition für die Funktion $type()$ ablesen:

$type(\Gamma, c)$ = der vorangegangene Typ für $c$

$$type(\Gamma, id) = \begin{cases} \Gamma(id), & \text{falls } id \in dom(\Gamma) \\ error, & \text{sonst} \end{cases}$$

$$type(\Gamma, e_1\, e_2) = \begin{cases} \tau', & \text{falls } type(\Gamma, e_1) = \tau \to \tau' \text{ und } type(\Gamma, e_2) = \tau \\ error, & \text{sonst (d. h. wenn } type(\Gamma, e_1) \text{ kein Funktionstyp)} \\ & \text{oder } type(\Gamma, e_2) \text{ nicht der dazu passende Argumenttyp ist} \\ & \text{oder wenn eins von beiden schon } error \text{ liefert)} \end{cases}$$

$$type(\Gamma, \boldsymbol{\lambda} id : \tau . e) = \begin{cases} \tau \to \tau', & \text{falls } type(\Gamma[\tau/id], e) = \tau' \\ error, & \text{sonst (d. h. wenn } type(\Gamma[\tau/id], e) = error) \end{cases}$$

usw. (**if then else**, **let**, **rec**: siehe Übung!)

Diese Definition lässt sich leicht in einen Algorithmus zur Typüberprüfung – d. h. zur Berechnung der Funktion $type()$ – umsetzen. Die Laufzeit des Algorithmus ist linear zur Größe von $e$.

#### Ist $\mathcal{L}_2^t$ noch typsicher?

DAZU: Übertragung der small step Semantik auf $\mathcal{L}_2^t$.

IDEE: Die vom Programmierer angegebenen Typen spielen zur Laufzeit keine Rolle, d. h.:

1. Entweder ändert man (BETA-V) und (UNFOLD) so ab, dass die Typen in den Ausdrücken stehen, aber ignoriert werden:
   (BETA-V') $(\boldsymbol{\lambda} id : \tau . e) v \to e[v/id]$
   (UNFOLD') $\mathbf{rec}\ id : \tau . e \to e[\mathbf{rec}\ id : \tau . e / id]$
2. Oder man definiert eine Funktion $erase()$, die die Typen aus einem Ausdruck $e \in \mathcal{L}_2^t$ entfernt und definiert die Semantik von $e$ als Semantik von $erase(e)$.

Die beiden Ansätze liefern im wesentlichen dasselbe: Wir erhalten Typsicherheit für $\mathcal{L}_2^t$.

BEWEISSKIZZE: Wenn $e :: \tau$ (in $\mathcal{L}_2^t$), dann ist auch $erase(e) :: \tau$ (in $\mathcal{L}_2$), denn aus einer Typherleitung für $e$ erhält man leicht eine für $erase(e)$. Also bleibt die Berechnung für $erase(e)$ nicht stecken, und damit auch die für $e$ nicht!

### 10.3 Typinferenz

IDEE: Der Programmierer darf Typen ins Programm schreiben, darf sie aber auch weglassen.

FORMAL: neue Sprache $\mathcal{L}_2^{ti}$ („$\mathcal{L}_2$ mit Typinferenz“)

Vorgegeben sei eine unendliche Menge $TVar$ sogenannter Typvariablen $\alpha$ $(\beta, \gamma, \dots)$[10].
Die Menge $Type$ aller Typen $\tau$ wird dann neu definiert, indem man folgende Produktion hinzunimmt:

$$\tau \quad ::= \quad \alpha$$

**Beispiel 30:** *(für neuartige Typen)*
$\alpha, \quad \alpha \to \beta, \quad \alpha \to \mathbf{int}, \quad \alpha \to \alpha \to \alpha, \quad \dots$

[10] In *SML*, *O'Caml*, *TPML*: *'a*,*'b*, ...

Implizit verändern sich damit auch die Typumgebungen $\Gamma$: Sie dürfen auch die neuartigen Typen enthalten, z. B. $\Gamma = [f : \alpha \to \alpha, x : \alpha, y : \mathit{int}]$
Die Menge *Exp* aller Ausdrücke von $\mathcal{L}_2^{ti}$ wird so definiert, dass $\boldsymbol{\lambda}$-Abstraktion und Rekursion jeweils mit und ohne Typ erlaubt sind, also

$$\begin{array}{rcl} e & ::= & \boldsymbol{\lambda} id.e_1 \\ & | & \boldsymbol{\lambda} id : \tau.e_1 \\ & | & \mathbf{rec}\ id.e_1 \\ & | & \mathbf{rec}\ id : \tau.e_1 \end{array}$$

**Beispiel 31:**

1. $\boldsymbol{\lambda} f.\boldsymbol{\lambda} x : \mathit{int}.f\,(f\,x)$
2. $\boldsymbol{\lambda} f : \alpha \to \alpha.\boldsymbol{\lambda} x : \alpha.f\,(f\,x)$
3. $\boldsymbol{\lambda} f : \alpha \to \mathit{int}.\boldsymbol{\lambda} x : \alpha.f\,(f\,x)$
4. $\boldsymbol{\lambda} f.\boldsymbol{\lambda} x.f\,(f\,x)$

**Typregeln für $\mathcal{L}_2^{ti}$:**

Für Abstraktion und Rekursion sind jetzt jeweils beide bisherigen Regeln erlaubt, also die 4 Regeln (Abstr), (T-Abstr), (Rec), (T-Rec).

**Small step Semantik für $\mathcal{L}_2^{ti}$:**

Wie für $\mathcal{L}_2^t$, d. h. die Semantik von $e \in \mathcal{L}_2^{ti}$ wird definiert als die Semantik von $\mathit{erase}(e) \in \mathcal{L}_2$.
Daraus folgt wie für $\mathcal{L}_2^t$ auch die Typsicherheit für $\mathcal{L}_2^{ti}$.

**Aufgabe des Typinferenzalgorithmus:**

- überprüfen, ob ein Ausdruck wohlgetypt ist
- wenn ja, einen „allgemeinsten“ Typ des Ausdrucks angeben

Ansatz: Man versucht eine Rückwärtsherleitung, bei der man das Raten vermeidet, indem man Typvariablen als „Platzhalter“ für noch nicht bekannte Typen benutzt.

**Beispiel 32:** *(Ausdruck (4) aus den Beispielen)*

Ziel: (1) $[\,] \rhd \boldsymbol{\lambda} f.\boldsymbol{\lambda} x.f\,(f\,x) :: \alpha_0$

Es kommt nur (Abstr) in Frage, aber nur unter der Einschränkung (engl.: *constraint*), dass $\alpha_0$ für einen Funktionstyp $\tau \to \tau'$ steht. Für die noch unbekannten Typen $\tau$,$\tau'$ wählen wir neue Typvariablen $\alpha_1$,$\alpha_2$. Also:

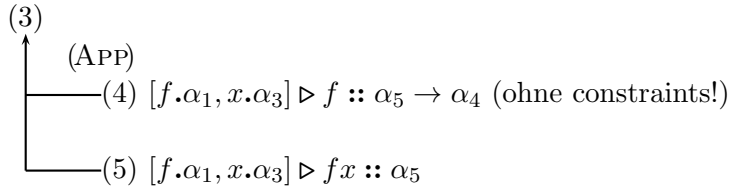
(1)
↑ (Abstr)
(2) $[f : \alpha_1] \rhd \boldsymbol{\lambda} x.f\,(f\,x) :: \alpha_2$ mit der Einschränkung (1a) $\alpha_0 = \alpha_1 \to \alpha_2$

Analog mit (2):

(2)
↑ (Abstr)
(3) $[f.\alpha_1, x.\alpha_3] \rhd f\,(f\,x) :: \alpha_4$ mit der Einschränkung (2a) $\alpha_2 = \alpha_3 \to \alpha_4$

BEACHTE: Die bisherigen Typgleichungen oder constraints (1a) und (2a) bringen nur zum Ausdruck, wie die Ergebnistypen am Ende „nach oben gereicht“ werden.

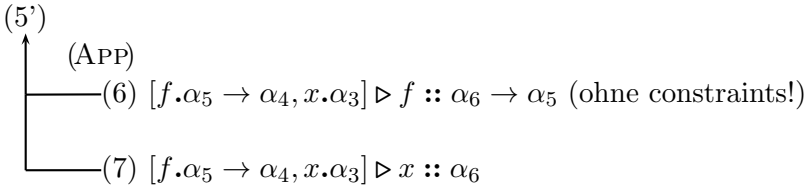
(3) kann nur mit (APP) behandelt werden. Dazu muss die linke Seite der Applikation – also $f$ – einen Funktionstyp mit Resultattyp $\alpha_4$ haben, also einen Typ der Form $\alpha_5 \to \alpha_4$ und die rechte Seite (d. h. $f\,x$) den zugehörigen Argumenttyp $\alpha_5$.




(4) ergibt sich mit (ID), wenn der constraint (4a) $\alpha_1 = \alpha_5 \to \alpha_4$ gilt.

Mit Gleichung (4a) kann man (5) umformen zu
(5') $[f.\alpha_5 \to \alpha_4, x.\alpha_3] \rhd fx :: \alpha_5$

(5') wird analog zu (3) behandelt:




(6) und (7) ergeben sich mit (ID) aus den constraints
(6a) $\alpha_5 \to \alpha_4 = \alpha_6 \to \alpha_5$
(7a) $\alpha_3 = \alpha_6$

Das Ziel (1) wurde zurückgeführt auf die Gleichungen (1a), (2a), (4a), (6a), (7a), die noch „gelöst“ werden müssen.

WAS HEISST „LÖSEN“?: $\Rightarrow$ Einsetzungen für die Typvariablen finden, sodass linke und rechte Seite der 5 Gleichungen syntaktisch übereinstimmen.

Aus (6a) und (7a) folgt, dass $\alpha_3 = \alpha_4 = \alpha_5 = \alpha_6$ gelten muss, d. h. dass die 4 Platzhalter für den gleichen Typ $\tau$ stehen. $\rightsquigarrow$ Wähle $\tau = \alpha$.

Aus (4a) folgt dann: $\alpha_1 = \alpha \to \alpha$
Aus (2a) folgt dann: $\alpha_2 = \alpha \to \alpha$
Aus (1a) folgt dann: $\alpha_0 = (\alpha \to \alpha) \to (\alpha \to \alpha)$
Also ist bewiesen:
$[\,]\boldsymbol{\lambda} f.\boldsymbol{\lambda} x.f\,(f\,x) :: \underbrace{(\alpha \to \alpha) \to (\alpha \to \alpha)}_{\text{allgemeinster Typ des Ausdrucks}}$

JETZT: Exakte Formulierung des Algorithmus. Dazu müssen zunächst einige Begriffe (Substitution, Anwendung einer Substitution, Lösung einer Gleichung, ...) eingeführt werden.

**Definition 16:** Eine (Typ-) Substitution ist eine totale Funktion $s : \mathit{TVar} \to \mathit{Type}$, für die gilt: Es existieren nur endlich viele $\alpha \in \mathit{TVar}$ mit $s(\alpha) \neq \alpha$.

SCHREIBWEISE: $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ steht für die Substitution $s : \mathit{TVar} \to \mathit{Type}$ mit
$s(\alpha_i) = \tau_i$ für $i = 1, \ldots, n$ und
$s(\alpha) = \alpha$ für alle $\alpha \notin \alpha_1, \ldots, \alpha_n$

*Subst* ist die Menge aller Substitutionen.

**Definition 17:** *(Anwendung einer Substitution auf einen Typ)*
Sei $\alpha, \tau \in \mathit{Type}, s \in \mathit{Subst}$.
Der Typ $\tau\, s$, der sich durch Anwendung von $s$ auf $\tau$ ergibt, ist induktiv definiert durch:

1. $\tau\, s = \tau$ falls $\tau \in \{\mathbf{int}, \mathbf{bool}, \mathbf{unit}\}$
2. $\alpha\, s = s(\alpha)$
3. $(\tau_1 \to \tau_2)\, s = \tau_1\, s \to \tau_2\, s$

**Beispiel 33:** $\underbrace{(\alpha_0 \to \mathbf{int} \to \alpha_1)}_{\tau}\underbrace{[\mathbf{int} \to \alpha_1/\alpha_0, \mathbf{bool}/\alpha_1]}_{s} = (\mathbf{int} \to \alpha_1) \to \mathbf{int} \to \mathbf{bool}$

Beachte: Es wird „simultan“ substituiert, nicht „nacheinander“, d. h. das neu entstehende $\alpha_1$ wird nicht durch $\mathbf{bool}$ ersetzt.

**Definition 18:** Seien $s_1$, $s_2 \in \mathit{Subst}$. Dann ist die Komposition $s_1\, s_2$ (oder: $s_1 \circ s_2$) definiert als die Funktion:

$$s_1 s_2 : \quad \mathit{TVar} \to \mathit{Type}$$
$$(s_1\, s_2)(\alpha) = \alpha\,(s_1\, s_2) = (s_1(\alpha))\, s_2 = (\alpha\, s_1)\, s_2$$

**Beispiel 34:** $\underbrace{[\beta/\alpha]}_{s_1} \circ \underbrace{[\mathbf{int}/\beta]}_{s_2} = \underbrace{[\mathbf{int}/\alpha, \mathbf{int}/\beta]}_{s_1 s_2}$

denn: $(\alpha\, s_1)\, s_2 = \beta\, s_2 = \mathbf{int}$
$(\beta\, s_1)\, s_2 = \beta\, s_2 = \mathbf{int}$
$(\alpha'\, s_1)\, s_2 = \alpha'\, s_2 = \alpha'$ für alle $\alpha' \notin \{\alpha, \beta\}$

**Lemma 11:** *Für alle $\tau \in \mathit{Type}$ und $s_1, s_2 \in \mathit{Subst}$ gilt: $\tau(s_1\, s_2) = (\tau\, s_1)\, s_2$*

**Beweis:** Gilt per Definition für Typvariablen $\alpha$.
Induktion $\rightsquigarrow$ für beliebige Typen $\tau$. □

Folgerung: Man schreibt einfach $\tau\, s_1\, s_2$ ohne Klammern.

**Lemma 12:** $(\mathit{Subst}, \circ)$ *ist ein Monoid mit neutralem Element* $[\,]$, *also gilt für alle* $s, s_1, s_2, s_3 \in \mathit{Subst}$:

1. $(s_1\, s_2)\, s_3 = s_1\,(s_2\, s_3)$ *(Assoziativität)*
2. $[\,] \circ s = s \circ [\,] = s$ *(Links- und Rechtsneutralität)*

**Beweis:** Da es sich um Gleichungen zwischen Funktionen handelt ist zu zeigen, dass die Funktion auf allen Typvariablen übereinstimmt, z. B.
(2) $\alpha(s \circ [\,]) = (\alpha\, s)\,[\,] = \alpha\, s$ für alle $\alpha$, also $s \circ [\,] = s$. □

Folgerung: Komposition zwischen mehreren Substitutionen darf auch ohne Klammern geschrieben werden, z. B. $s_1\, s_2\, s_3\, s_4$.

**Definition 19:** Eine Typgleichung ist eine „Formel“ der Gestalt $\tau_1 = \tau_2$. Eine solche Gleichung heißt gültig, wenn $\tau_1$ und $\tau_2$ syntaktisch gleich sind.

**Definition 20:** Sei $s \in \mathit{Subst}$.

1. Für eine Typgleichung $eq$ der Form $\tau_1 = \tau_2$ bezeichnet $eq\, s$ die Gleichung $\tau_1\, s = \tau_2\, s$.
   $s$ heißt Lösung von $eq$, wenn $eq\, s$ gültig ist.
2. Für eine Menge $E$ von Typgleichungen sei $E\, s = \{eq\, s \mid eq \in E\}$.
   $s$ heißt Lösung von $E$, wenn $E\, s$ nur aus gültigen Gleichungen besteht.

**Beispiel 35:** Mögliche Lösungen für $\alpha \to \beta = \beta \to \alpha$ sind:
$s_1 = [\beta/\alpha] \rightsquigarrow \beta \to \beta = \beta \to \beta$
$s_2 = [\alpha/\beta] \rightsquigarrow \alpha \to \alpha = \alpha \to \alpha$
$s_3 = [\mathit{int}/\alpha, \mathit{int}/\beta] \rightsquigarrow \mathit{int} \to \mathit{int} = \mathit{int} \to \mathit{int}$

BEMERKUNGEN: $s_1$ und $s_2$ sind „allgemeine Lösungen“, $s_3$ ist eine “zu spezielle“ Lösung, da $\alpha$ und $\beta$ unnötigerweise auf ***int*** festgelegt werden.

**Definition 21:** Seien $s_1$,$s_2 \in \mathit{Subst}$.
$s_1$ heißt allgemeiner als $s_2$ (Schreibweise: $s_1 \sqsubseteq s_2$), wenn es eine Substitution $s$ gibt mit $s_1\, s = s_2$.

INTUITION: Indem man „nach“ $s_1$ eine weitere Substitution $s$ ausführt, erhält man insgesamt eine spezielle Substitution $s_2$.

**Beispiel 36:** $[\beta/\alpha] \sqsubseteq [\mathit{int}/\alpha, \mathit{int}/\beta]$, denn $[\beta/\alpha] \circ [\mathit{int}/\beta] = [\mathit{int}/\alpha, \mathit{int}/\beta]$.

**Lemma 13:** *$(\mathit{Subst}, \sqsubseteq)$ ist eine Präordnung, d. h. für alle $s$, $s_1$, $s_2$, $s_3 \in \mathit{Subst}$ gilt:*

1. *$s \sqsubseteq s$ (Reflexivität)*
2. *$s_1 \sqsubseteq s_2$ und $s_2 \sqsubseteq s_3 \Rightarrow s_1 \sqsubseteq s_3$ (Transitivität)*
3. *$[\,] \sqsubseteq s$ für alle $s \in \mathit{Subst}$, $[\,]$ ist also die Allgemeinste unter allen Substitutionen.*

**Beweis:** klar □

**Beispiel 37:** $[\beta/\alpha] \sqsubseteq [\alpha/\beta]$, denn $[\beta/\alpha] \circ [\alpha/\beta] = [\alpha/\beta]$. Umgekehrte Ungleichung analog.

BEACHTE: $\sqsubseteq$ ist keine (partielle) Ordnung, denn es fehlt die Antisymmetrie, d. h. es kann $s_1 \sqsubseteq s_2$ und $s_2 \sqsubseteq s_1$ gelten, ohne dass $s_1 = s_2$.

**Beispiel 38:** $s_1 = [\beta/\alpha], s_2 = [\alpha/\beta]$
Es gilt: $s_1\, s_2 = [\alpha/\beta] = s_2 \rightsquigarrow s_1 \sqsubseteq s_2$
Aber: $s_2\, s_1 = [\beta/\alpha] = s_1 \rightsquigarrow s_2 \sqsubseteq s_1$

**Definition 22:** Sei $s \in \mathit{Subst}$.
$s$ heißt allgemeinste Lösung einer Typgleichung $eq$ (bzw. Gleichungsmenge $E$), wenn gilt:

1. $s$ ist eine Lösung von $eq$ (bzw. von $E$)
2. $s \sqsubseteq s'$ für jede Lösung $s'$ von $eq$ (bzw. $E$), d. h. $s$ ist allgemeiner als jede (andere) Lösung

**Beispiel 39:** Eine allgemeinste Lösung der Gleichung $\alpha \to \beta = \beta \to \alpha$ ist $s = [\beta/\alpha]$, denn:

1. $s$ ist Lösung $\surd$
2. Sei $s'$ Lösung dieser Gleichung, d. h. $s'(\alpha) \to s'(\beta) = s'(\beta) \to s'(\alpha)$,
   also $s'(\alpha) = s'(\beta) \rightsquigarrow$ betrachte $s\, s'$
   Es gilt $\alpha\, s\, s' = \beta\, s' = s'(\beta) = s'(\alpha) = \alpha\, s'$, also $\alpha'\, s\, s' = s'\, \alpha'$ für $\alpha' \neq \alpha$.

BEACHTE: Eine Gleichung(smenge) muss keine Lösung haben, z. B.:

1. ***int*** $=$ ***bool***
2. $\alpha = \alpha \to$ ***int*** (Für jede Substitution $s$ gilt: $s(\alpha)$ und $s(\alpha) \to$ ***int*** sind nicht syntaktisch gleich, d. h. es gibt keine Lösung $s$.)

WIR WERDEN SEHEN: Wenn irgendeine Lösung existiert, dann existiert sogar eine allgemeinste Lösung (die man mit dem Unifikationsalgorithmus findet).

BEACHTE: Wenn man eine allgemeinste Lösung $s$ kennt, dann kennt man alle Lösungen, denn:
$s$ ist allgemeinste Lösung $\Rightarrow \{s' \in \mathit{Subst} \mid s \sqsubseteq s'\}$ ist die Menge aller Lösungen. Wenn $s$ beide Seiten einer Gleichung syntaktisch gleich macht, dann gilt das erst recht für $s' = s \circ s''$.

## 10.4 Der Unifikations-Algorithmus

Der Algorithmus wird als rekursive Funktion *unify*() beschrieben, die als Argument eine Gleichungsmenge $E$ hat und als Ergebnis eine Substitution $s$ liefert (nämlich die allgemeinste Lösung von $E$). Der Algorithmus ist nichtdeterministisch: Solange die Gleichungsmenge nicht leer ist, wählt man eine Gleichung aus und trifft dann eine Fallunterscheidung nach der Form der Gleichung. Da sich die Fälle teilweise überlappen, spielt die Reihenfolge der Fälle keine Rolle.

(EMPTY) $\mathit{unify}(\emptyset) = [\,]$

(TRIV) $\mathit{unify}(\{\tau = \tau\} \cup E) = \mathit{unify}(E)$

(ARROW) $\mathit{unify}(\{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\} \cup E) = \mathit{unify}(\{\tau_1 = \tau_1', \tau_2 = \tau_2'\} \cup E)$

(VAR)
$$\underbrace{\mathit{unify}(\{\alpha = \tau\} \cup E)}_{=\mathit{unify}(\{\tau=\alpha\} \cup E)} = \begin{cases} [\tau/\alpha] \circ s, & \text{falls } s = \mathit{unify}(E[\tau/\alpha]) \\ & \text{und } \alpha \text{ nicht in } \tau \text{ vorkommt} \\ \text{„nicht lösbar“}, & \text{falls } \alpha \text{ in } \tau \text{ vorkommt oder} \\ & \mathit{unify}(E[\tau/\alpha]) = \text{„nicht lösbar“} \end{cases}$$

(STRUCT) $\mathit{unify}(\{\tau_1 = \tau_2\} \cup E) =$ „nicht lösbar“ (alle anderen Fälle)

**Beispiel 40:** $\mathit{unify}(\{\alpha \to \mathit{int} = \beta \to \alpha\})$

$\underset{\text{(ARROW)}}{=} \mathit{unify}(\{\alpha = \beta, \mathit{int} = \alpha\}) \underset{\text{(VAR)}}{=} [\beta/\alpha] \circ s$ mit $s = \mathit{unify}(\{\mathit{int} = \beta\})$

$\underset{\text{(VAR)}}{=} [\mathit{int}/\beta] \circ s_1$ mit $s_1 = \mathit{unify}(\emptyset) \underset{\text{(EMPTY)}}{=} [\,]$

also $s = [\mathit{int}/\beta] \circ [\,] = [\mathit{int}/\beta] = [\beta/\alpha] \circ [\mathit{int}/\beta] = [\mathit{int}/\alpha, \mathit{int}/\beta]$

### Korrektheit des Unifikationsalgorithmus:

Es ist zu zeigen:

1. Der Algorithmus terminiert immer.
2. Der Algorithmus liefert das gewünschte Ergebnis (sog. partielle Korrektheit), d. h.
   - „nicht lösbar“, falls keine Lösung für $E$ existiert
   - eine allgemeinste Lösung, falls eine Lösung existiert.

ZUNÄCHST (1): Um Terminieren einer rekursiven Funktion zu beweisen, zeigt man, dass das Argument der Funktion bei jedem rekursiven Aufruf in irgendeinem Sinne „kleiner“ oder „einfacher“ wird, wobei „kleiner“ so definiert sein muss, dass das Argument nicht unendlich oft „kleiner“ werden kann.

IM UNIFIKATIONSALGORITHMUS: Beim rekursiven Aufruf in Regel (VAR) wird man die Typvariable $\alpha$ los, denn $\alpha$ kommt in $\tau$ nicht vor, also auch nicht mehr im neuen Argument $E[\tau/\alpha]$ von *unify*().

ALSO: Bei (VAR) wird die Anzahl der Typvariablen in $E$ kleiner.
Da bei den anderen rekursiven Aufrufen keine Typvariablen hinzukommen, kann (VAR) also nur endlich oft aufgerufen werden.

$\rightsquigarrow$: Es genügt zu zeigen, dass man immer wieder zu (VAR) kommen muss, d. h. dass es keine endlose Rekursion ohne (VAR) gibt.
Dafür kommen nur noch (TRIV) und (ARROW) in Frage. $\Rightarrow$ Hier ist keine Endlos-Rekursion möglich, da bei beiden Regeln die Gesamtlänge des Gleichungssystems[11] kürzer wird.

[11] also die Summe der Länge aller Gleichungen

BEACHTE: Bei (VAR) kann die Gesamtlänge des Gleichungssystems „unkontrolliert“ wachsen, da $E[\tau/\alpha]$ größer als $E$ sein kann. Das schadet aber nicht beim Terminieren.

NOCH ZU ZEIGEN: *unify*$(E)$ liefert stets eine korrekte Antwort.

**Definition 23:**

- *var*$(\tau)$ ist die Menge aller Typvariablen in $\tau$.
- *var*$(E)$ ist die Menge aller Typvariablen in $E$.

**Lemma 14:** *(Allgemeinste Lösung von Typgleichungen)*

1. *Wenn $\alpha \notin var(\tau)$, dann ist die Substitution $[\tau/\alpha]$ eine allgemeinste Lösung der Gleichung $\alpha = \tau$ (oder $\tau = \alpha$).*
2. *Wenn $s_1$ eine allgemeinste Lösung von $E_1$ ist und $s_2$ eine allgemeinste Lösung von $E_2\, s_1$, dann ist $s_1\, s_2$ eine allgemeinste Lösung von $E_1 \cup E_2$.*

**Beweis:**

1. $[\tau/\alpha]$ ist eine Lösung von $\alpha = \tau$, denn $\alpha[\tau/\alpha] = \tau$
   $\tau[\tau/\alpha] = \tau$ (weil $\alpha \notin var(\tau)$)

   Sei $s$ eine (andere) Lösung von $\alpha = \tau$.
   Zu zeigen: $[\tau/\alpha] \sqsubseteq s$, d. h. es existiert ein $s'$ mit $[\tau/\alpha] \circ s' = s$
   Wir wählen $s' = s$. Dann gilt:
   $$\left.\begin{array}{lcl} \alpha[\tau/\alpha]s & = & \tau\, s = \alpha\, s \text{(weil } s \text{ Lösung von } \alpha = \tau) \\ \alpha'[\tau/\alpha]s & = & \alpha'\, s \text{ für alle } \alpha' \neq \alpha \end{array}\right\} \text{ also } [\tau/\alpha] \circ s = s \text{ d. h. } [\tau/\alpha] \sqsubseteq s$$
2. $s_1\, s_2$ ist Lösung von $E_1 \cup E_2$, denn:
   Da $s_1$ Lösung von $E_1$ ist, besteht $E_1\, s_1$ nur aus gültigen Gleichungen, also erst recht $E_1\, s_1\, s_2$ (denn die Anwendung einer Substitution auf eine gültige Gleichung liefert wieder eine gültige Gleichnung). Da $s_2$ Lösung von $E_2\, s_1$ ist, besteht $E_2\, s_1\, s_2$ nur aus gültigen Gleichungen, d. h. $s_1\, s_2$ ist eine Lösung von $E_2$.
   Zusammen: $s_1\, s_2$ ist eine Lösung von $E_1 \cup E_2$.

   Sei $s$ eine (andere) Lösung von $E_1 \cup E_2$.
   Da $s$ Lösung von $E_1$ ist, gilt $s_1 \sqsubseteq s$, d. h. es existiert $s_1\, s' = s$. Da $s$ Lösung von $E_2$ ist, besteht $E\, s_1\, s' = E\, s$ nur aus gültigen Gleichungen, d. h. $s'$ ist Lösung von $E_2\, s_1$. Also gilt $s_2 \sqsubseteq s'$, d. h. es existiert ein $s''$ mit $s_2 s'' = s'$.
   Zusammen: $s_1\, s_2\, s'' = s_1\, s' = s$, also $s_1\, s_2 \sqsubseteq s$. □

**Satz 11:** *(Totale Korrektheit des Unifikations-Algorithmus)*
*unify*$(E)$ *terminiert für jede endliche Gleichgungsmenge $E$ mit*

- *einer allgemeinsten Lösung $s$ von $E$, falls $E$ eine Lösung hat.*
- *der Antwort „nicht lösbar“, falls $E$ keine Lösung hat.*

**Beweis:** Terminierung √
Durch Induktion über die Rekursionstiefe (die wegen Terminierung endlich ist) zeigen wir, dass stets die richtige Antwort geliefert wird.

Rekursionstiefe 0: (d. h. kein rekursiver Aufruf)

(EMPTY) Jede Substitution $s$ ist Lösung von $E =$, also ist $[\,]$ allgemeinste Lösung. √

(VAR) greift im Fall $\alpha \in var(\tau)$ und $\alpha \neq \tau$.
Wenn $\alpha \in var(\tau)$, dann ist $\alpha s$ ein echter Teil von $\tau s$ für jede Substitution $s$, also existiert keine Lösung für $\alpha = \tau$. √

(STRUCT) betrifft Gleichungen $\tau_1 = \tau_2$, wobei

- $\tau_1$ und $\tau_2$ nicht syntaktisch gleich sind
- $\tau_1$ und $\tau_2$ beides Funktionstypen sind
- weder $\tau_1$ noch $\tau_2$ eine Typvariable ist.

$\leadsto$ Einer der beiden Typen muss ein Basistyp und der andere entweder ein Basistyp oder ein Funktionstyp sein.
$\leadsto$ Keine Substitution $s$ kann $\tau_1$ und $\tau_2$ gleich machen, also existiert keine Lösung. $\surd$

Rekursionstiefe $> 0$:

(TRIV) Es gilt: $s$ ist Lösung von $\{\tau = \tau\} \cup E \iff s$ ist Lösung von $E$ (da jede Substitution $\tau = \tau$ löst). Daraus folgt: $s$ ist allgemeinste Lösung von $\{\tau = \tau\} \cup E \iff s$ ist allgemeinste Lösung von $E$.
Also: Wenn $\{\tau = \tau\} \cup E$ keine Lösung hat, dann hat auch $E$ keine, also ist nach Induktionsannahme *unify*$(E)$ = „nicht lösbar“, und das ist die korrekte Antwort.
Andernfalls liefert *unify*$(E)$ nach Induktionsannahme eine allgemeinste Lösung von $E$, also auch die korrekte Antwort. $\surd$

(ARROW) Es gilt: $s$ ist Lösung von $\{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\} \cup E$
$\iff s$ ist Lösung von $\{\tau_1 = \tau_1', \tau_2 = \tau_2'\} \cup E$.
Weitere Argumentation wie bei (TRIV). $\surd$

(VAR) greift im Fall $\alpha \notin \mathit{var}(\tau)$. Nach dem Lemma ist $[\tau/\alpha]$ allgemeinste Lösung von $\alpha = \tau$. Wenn *unify*$(E[\tau/\alpha]) = s$, dann ist $s$ nach Induktionsannahme allgemeinste Lösung von $E[\tau/\alpha]$. Also ist nach Teil 2 des Lemmas $[\tau/\alpha] \circ s$ allgemeinste Lösung von $\{\alpha = \tau\} \cup E$, also ist die Antwort korrekt.
Wenn *unify*$(E[\tau/\alpha])$ = „nicht lösbar“, dann hat $E[\tau/\alpha]$ nach Induktionsannahme keine Lösung.
Wäre $s$ eine Lösung von $\{\alpha = \tau\} \cup E$, dann wäre $[\tau/\alpha] \sqsubseteq s$ (da $s$ eine Lösung von $\alpha = \tau$ ist), d. h. es existiert ein $s'$ mit $[\tau/\alpha] \circ s' = s$. Da $s$ eine Lösung von $E$ ist, besteht also $E[\tau/\alpha]\, s'$ nur aus gültigen Gleichungen, d. h. $s'$ wäre eine Lösung von $E[\tau/\alpha]$. Das ist ein Widerspruch. Also hat auch $\{\alpha = \tau\} \cup E$ keine Lösung. $\surd$ □

## 10.5 Der Typinferenz-Algorithmus

ZIEL: Allgemeinsten Typ eines Ausdrucks $e$ finden.

PRINZIP: Man versucht, eine Typherleitung für $[] \rhd e :: \alpha$ zu konstruieren. Dabei vermeidet man das Raten, indem man für noch unbekannte Typen stets neue Typvariablen einführt.

FORMAL: Wir definieren eine rekursive Funktion *solve*(), die zu Beginn mit dem Typurteil $[] \rhd e :: \alpha$ aufgerufen wird und eine „allgemeinste Lösung“ dafür liefert. Durch die Rekursion wird ein Typurteil durch Rückwärtsanwendung der Typregeln auf „Teilziele“ zurückgeführt, d. h. auf andere Typurteile und Typgleichungen ($\leadsto$ constraints).

Auch *solve*() ist – wie *unify*() – nichtdeterministisch: Solange $F$ nicht leer ist, kann man eine beliebige Formel aus $F$ auswählen. Nach dieser ausgewählten Formel wird folgende Fallunterscheidung getroffen:

- Wenn die Formel ein Typurteil ist, dann wird es mit der passenden Typregel auf andere Typurteile und/oder Gleichungen zurückgeführt.
- Wenn die ausgewählte Formel eine Gleichung ist, wird *unify*() aufgerufen[12] und die Lösung $s$ dieser Gleichung wird auf alle restlichen Formeln angewandt.

[12] Spezialfall: Die entstehenden Gleichungen löst man erst am Ende.

**Lösung von Typurteilen:**

**Definition 24:** *(Anwendung einer Substitution auf Typurteile)*
Sei $s \in Subst$.

- Sei $e \in Exp$. Dann bezeichnet $e\,s$ den Ausdruck, der aus $e$ entsteht, indem man jeden Typ $\tau$, der in $e$ vorkommt, duch $\tau\,s$ ersetzt.
- Sei $\Gamma \in TEnv$. Dann ist $\Gamma\,s$ die Typumgebung, die aus $\Gamma$ entsteht, indem man jeden Eintrag $id : \tau$ durch $id : \tau\,s$ ersetzt.
  Formal:
  $\Gamma\,s : Id \hookrightarrow Type$
  $dom(\Gamma s) = dom(\Gamma)$
  $\Gamma\,s(id) = \Gamma(id)\,s$
- Sei $tj$ ein Typurteil $\Gamma \rhd e :: \tau$. Dann ist $tj\,s$ das Typurteil $\Gamma s \rhd e\,s :: \tau s$.
- Sei $F$ eine (endliche) Menge von „Formeln“ $f$, d. h. von Typurteilen und Gleichungen. Dann bezeichnet $F\,s$ die Menge $\{f s \mid f \in F\}$.

**Definition 25:**

1. $s$ heißt Lösung des Typurteils $\Gamma \rhd e :: \tau$, falls $\Gamma s \rhd e s :: \tau s$ gültig (also mit den Typregeln herleitbar) ist.
2. $s$ heißt Lösung einer Formelmenge $F$, wenn die Menge $F\,s$ nur aus gültigen Formeln (Typurteilen und Gleichungen) besteht.
3. $s$ heißt allgemeinste Lösung eines Typurteils $tj$ oder einer Formelmenge $F$, wenn $s$ eine Lösung und allgemeiner als jede andere Lösung ist.

**Beispiel 41:** Sei $tj$ das Typurteil $[f : \alpha_2] \rhd \boldsymbol{\lambda} x : \alpha_1 . f\,(f\,x) :: \alpha_0$
Mögliche Lösungen von $tj$ sind:

1. $s_1 = [\alpha_1 \to \alpha_1/\alpha_2, \alpha_1 \to \alpha_1/\alpha_0]$
   denn $tj\,s_1$ ist das gültige Typurteil $[f : \alpha_1 \to \alpha_1] \rhd \boldsymbol{\lambda} x : \alpha_1 . f\,(f\,x) :: \alpha_1 \to \alpha_1$
2. $s_2 = [\mathbf{int} \to \mathbf{int}/\alpha_2, \mathbf{int}/\alpha_1, \mathbf{int} \to \mathbf{int}/\alpha_0]$
   denn $tj\,s_2$ ist das gültige Typurteil $[f : \mathbf{int} \to \mathbf{int}] \rhd \boldsymbol{\lambda} x : \mathbf{int} . f(f\,x) :: \mathbf{int} \to \mathbf{int}$

Es gilt $s_1 \sqsubseteq s_2$, denn $s_1 \circ [\mathbf{int}/\alpha_1] = s_2$ und es lässt sich zeigen, dass $s_1$ die allgemeinste Lösung von $tj$ ist.

**Regeln des Typinferenzalgorithmus:**

(Empty) $solve(\emptyset) = [\,]$

(Const) $solve(\{\Gamma \rhd c :: \tau\} \cup F) = solve(\{\tau = \tau'\} \cup F)$ falls $c :: \tau'$

(Id) $$solve(\{\Gamma \rhd id :: \tau\} \cup F) = \begin{cases} \text{„nicht lösbar“} & \text{falls } id \in dom(\Gamma) \\ solve(\{\tau = \tau'\} \cup F) & \text{falls } id \in dom(\Gamma) \text{ und } \Gamma(id) = \tau' \end{cases}$$

(App) $solve(\{\Gamma \rhd e_1 e_2 :: \tau\} \cup F) = solve(\{\Gamma \rhd e_1 :: \alpha \to \tau, \Gamma \rhd e_2 :: \alpha\} \cup F)$ wobei $\alpha$ neue Typvariable

(Cond) $solve(\{\Gamma \rhd \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 :: \tau\} \cup F) = solve(\{\Gamma \rhd e_0 :: \mathbf{bool}, \Gamma \rhd e_1 :: \tau, \Gamma \rhd e_2 :: \tau\} \cup F)$

(Abstr) $solve(\{\Gamma \rhd \boldsymbol{\lambda} id . e :: \tau\} \cup F) = solve(\{\Gamma[\alpha'/id] \rhd e :: \alpha, \tau = \alpha' \to \alpha\} \cup F)$

(T-Abstr) $solve(\{\Gamma \rhd \boldsymbol{\lambda} id : \tau' . e :: \tau\} \cup F) = solve(\{\Gamma[\tau'/id] \rhd e :: \alpha, \tau = \tau' \to \alpha\} \cup F)$

(Rec) $solve(\{\Gamma \rhd \mathbf{rec}\ id.e :: \tau\} \cup F) = solve(\{\Gamma[\tau/id] \rhd e :: \tau\} \cup F)$

(T-Rec) $solve(\{\Gamma \rhd \mathbf{rec}\ id : \tau'.e :: \tau\} \cup F) = solve(\{\Gamma[\tau/id] \rhd e :: \tau, \tau = \tau'\} \cup F)$

(Unify) $solve(\{\tau_1 = \tau_2\} \cup F) = \begin{cases} s_1 s_2 & \text{falls } s_1 = unify(\{\tau_1 = \tau_2\}) \text{ und } s_2 = solve(F\, s_1) \\ \text{„nicht lösbar“} & \text{sonst} \end{cases}$

**Beispiel 42:** $solve(\{[\,] \rhd (\boldsymbol{\lambda} x.x)1 :: \alpha_0\})$

$= \underset{(App)}{\{[\,] \rhd \boldsymbol{\lambda} x.x :: \alpha_1 \to \alpha_0 [\,] \rhd 1 :: \alpha_1\}}$

$= \underset{(Abstr)}{\{[x :: \alpha_2] \rhd x :: \alpha_3, \alpha_2 \to \alpha_3 = \alpha_1 \to \alpha_0, [\,] \rhd 1 :: \alpha_1\}}$

$= \underset{(Id)}{\{\alpha_2 = \alpha_3, \alpha_2 \to \alpha_3 = \alpha_1 \to \alpha_0, [\,] \rhd 1 :: \alpha_1\}}$

$= \underset{(Unify)}{[\alpha_2/\alpha_3] \circ solve(\{\alpha_2 \to \alpha_2 = \alpha_1 \to \alpha_0, [\,] \rhd 1 :: \alpha_1\})}$

$= \underset{(Unify)}{[\alpha_2/\alpha_3] \circ [\alpha_2/\alpha_0, \alpha_2/\alpha_1] \circ solve(\{[\,] \rhd 1 :: \alpha_2\})}$

$= \underset{(Const)}{[\alpha_2/\alpha_3] \circ [\alpha_2/\alpha_0, \alpha_2/\alpha_1] \circ solve(\{\alpha_2 = \mathit{int}\})}$

$= \underset{(Unify)}{[\alpha_2/\alpha_3] \circ [\alpha_2/\alpha_0, \alpha_2/\alpha_1] \circ [\mathit{int}/\alpha_2] \circ solve()}$

$= \underset{(Empty)}{[\alpha_2/\alpha_3] \circ [\alpha_2/\alpha_0, \alpha_2/\alpha_1] \circ [\mathit{int}/\alpha_2] \circ [\,]}$

$= [\mathit{int}/\alpha_0, \mathit{int}/\alpha_1, \mathit{int}/\alpha_2, \mathit{int}/\alpha_3]$ Es interessiert nur $solve(\alpha_0) = \mathit{int}$.
Durch „Lösen“ des ursprünglichen Typurteils entsteht das gültige Typurteil $[\,] \rhd (\boldsymbol{\lambda} x.x)\, 1 :: \mathit{int}$

Beachte: Der Algorithmus liefert eine Substitution $s$, die nicht die allgemeinste Lösung des ursprünglichen Typurteils ist.
Erst Einschränkung von $s$ auf die urprünglichen Typvariablen (im Beispiel $\alpha_0$) ist die allgemeinste Lösung (also im Beispiel die Substitution $[\mathit{int}/\alpha_0]$).

### Allgemeinster Typ eines Ausdrucks:

Der Programmierer entscheidet:

- Ist der eingegebene Ausdruck wohlgetypt?
- Wenn ja, wie sieht der „allgemeinste Typ“ aus?

**Definition 26:** Seien $\tau, \tau' \in Type$

1. $\tau$ heißt allgemeiner als $\tau'$ (Schreibweise: $\tau \sqsubseteq \tau'$), wenn ein $s \in Subst$ existiert, so dass $\tau\, s = \tau'$.
2. Sei $\Gamma \in TEnv$ und $e \in Exp$ ein Ausdruck ohne Typvariablen.
   $\tau$ heißt allgemeinster Typ von $e$ bzgl. $\Gamma$, wenn gilt:
   - $\Gamma \rhd e :: \tau$ ist gültig
   - für jeden Typ $\tau'$ mit $\Gamma \rhd e :: \tau'$ gültig folgt $\tau \sqsubseteq \tau'$

   Wenn $e$ abgeschlossen ist, dann sagt man „allgemeinster Typ von $e$“ statt „allgemeinster Typ von $e$ bzgl. $[\,]$“.

**Beispiel 43:**

zu (1):

- $\alpha \to \alpha \sqsubseteq \mathit{int} \to \mathit{int}$
- $\alpha_1 \to \alpha_2 \sqsubseteq \mathit{int} \to \mathit{int}$
- Jedes $\alpha \in \mathit{TVar}$ ist allgemeinster Typ unter allen. ($\rightsquigarrow$ Auch $(\mathit{Type}, \sqsubseteq)$ ist eine Präordnung mit kleinstem Element.)

zu (2):

- $\boldsymbol{\lambda} x.x$ hat allgemeinsten Typ $\alpha \to \alpha$
- $\boldsymbol{\lambda} x.1$ hat allgemeinsten Typ $\alpha \to \mathit{int}$
- $\boldsymbol{\lambda} f.\boldsymbol{\lambda} x.f\,(f\,x)$[13] hat allgemeinsten Typ $(\alpha \to \alpha) \to (\alpha \to \alpha)$
- $\mathbf{rec}\ x.x$ hat allgemeinsten Typ $\alpha$
- $[\,]$ hat allgemeinsten Typ $\alpha$ **list**
- $hd\,[\,]$ hat allgemeinsten Typ $\alpha$

Es gilt: Wenn $\Gamma$ und $e$ keine Typvariablen enthalten, dann liefert $\mathit{solve}(\{\Gamma \rhd e :: \alpha_0\})$ eine Substitution $s$ so, dass $s(\alpha_0)$ allgemeinster Typ von $e$ in $\Gamma$ ist, falls überhaupt ein Typ von $e$ in $\Gamma$ existiert, ansonsten liefert er „nicht lösbar“.

Auf diese Weise arbeitet der Compiler von *SML* bzw. *O'Caml*: Durch Deklarationen ist bereits eine Typumgebung $\Gamma$ vorgegeben. Dann gibt man einen Ausdruck $e$ ein und erhält vom Compiler einen allgemeinsten Typ für $e$ in $\Gamma$ (oder eine Fehlermeldung, falls kein Typ existiert).

### Korrektheit des Typinferenzalgorithmus:

Schreibweise: Sei $s \in \mathit{Subst}$ und $A \subset \mathit{TVar}$.
Dann sei $s\,|_A$ die Einschränktung von $s$ auf $A$, definiert als die Substitution
$s' : \mathit{TVar} \to \mathit{Type}$ mit
$s'(\alpha) = s(\alpha)$ für alle $\alpha \in A$
$s'(\alpha) = \alpha$ für alle $\alpha \notin A$

**Satz 12:** *(Totale Korrektheit des Typinferenzalgorithmus)*
*$\mathit{solve}(F)$ terminiert für jede endliche Formelmenge $F$, wobei gilt:*

- *Wenn $F$ eine Lösung besitzt und $A$ die Menge aller Typvariablen in $F$ ist, dann liefert $\mathit{solve}(F)$ eine Substitution $s$, deren Einschränkung $s\,|_A$ die allgemeinste Lösung von $F$ ist.*
- *Wenn $F$ keine Lösung besitzt, dann liefert $\mathit{solve}(F)$ die Antwort „nicht lösbar“.*

**Beweis:** Terminieren: klar, da durch die Regeln für Typurteile die Ausdrücke stets kleiner werden und andererseits die Regel (unify) dabei nicht stört.

Partielle Korrektheit: ohne Beweis. □

**Korollar 2:** *Wenn $e \in \mathit{Exp}$ ohne Typvariablen, dann terminiert $\mathit{solve}(\{[\,] \rhd e :: \alpha\})$, wobei gilt:*

- *Wenn $e$ einen Typ besitzt, dann liefert $\mathit{solve}(\{[\,] \rhd e :: \alpha\})$ eine Substitution $s$ so, dass $s(\alpha)$ allgemseinster Typ von $e$ ist.*
- *Wenn $e$ keinen Typ besitzt, dann liefert $\mathit{solve}(\{[\,] \rhd e :: \alpha\})$ die Antwort „nicht lösbar“.*

[13] steht für $\mathit{twice}(f) = f \circ f$

**Beweis:** Folgt sofort aus dem Satz, da $A = \{\alpha\}$:
$solve(\{[\,] \triangleright e :: \alpha\})$ liefert eine Substitution $s$, deren Einschränkung $s\,|_A$ allgemeinste Lösung ist. $\Rightarrow s(\alpha)$ ist der allgemeinste Typ von e. □

BEMERKUNG: Auch für kleine Ausdrücke können sich große Typen ergeben.

**Beispiel 44:** Sei $double = \boldsymbol{\lambda} x.\boldsymbol{\lambda} f.f(fx)$.
$\rightsquigarrow$ *double* hat allgemeinsten Typ $\alpha \to (\alpha \to \alpha) \to \alpha$.
$\Rightarrow$ Wenn $e :: \tau$, dann ist $double\ e :: (\tau \to \tau) \to \tau$.
$\Rightarrow \underbrace{double(\ldots(double\,1)\ldots)}_{n}$ hat Typ der Größenordnung $3^n$

# 11 Polymorphie

Was ist eine polymorphe Funktion?

DUDEN: polymorph = „vielgestaltig“

Eine polymorphe Funktion ist eine Funktion, die unterschiedliche Typen annehmen kann ($\rightsquigarrow$ sie kann auf Argumente unterschiedlichen Typs angewandt werden und Resultate unterschiedlichen Typs liefern).

SPEZIELL: parametrische Polymorphie
Man hat einen oder mehrere Typparameter $\alpha, \beta$. Für jede Instanziierung der Typparameter durch „konkrete“ Typen erhält man eine Funktion, alle diese Funktionen sind „gleichartig“.

## 11.1 Haben wir durch die Typvariablen Polymorphie eingeführt?

**Beispiel 45:**

- $\boldsymbol{\lambda} x.x$ kann Typ $\tau \to \tau$ für jeden Typ $\tau$ annehmen
- $\boldsymbol{\lambda} f.\boldsymbol{\lambda} x.f\,(f\,x)$ kann Typ $(\tau \to \tau) \to (\tau \to \tau)$ für jedes $\tau \in \mathit{Type}$ annehmen
- Listenfunktionen wie *hd*,*tl*,*map*,*member*, ... sind polymorph, z. B.

  *hd*: $\tau$ **list** $\to \tau$ für jedes $\tau$

  *tl*: $\tau$ **list** $\to \tau$ **list** für jedes $\tau$

  *map*: $(\tau \to \tau') \to \tau$ **list** $\to \tau'$ **list** für alle $\tau, \tau' \in \mathit{Type}$

  *member*: $\tau \to \tau$ **list** $\to$ ***bool*** für alle $\tau \in \mathit{Type}$

In unserer Sprache $\mathcal{L}_2^{ti}$ (oder $\mathcal{L}_3^{ti}$ mit Listen und Tupeln) gibt es nur polymorphe Konstanten wie [ ], *hd*, *tl*, aber keine benutzerdefinierte Polymorphie.

**Beispiel 46:** **let** $f = \boldsymbol{\lambda} x.x$ **in if** $f$ *true* **then** $f\,1$ **else** $f\,2$ terminiert mit 1.
aber: Es ist nicht wohlgetypt in $\mathcal{L}_2^{ti}$, denn:
Für den Namen $f$ wird zwar der (scheinbar polymorphe) Typ $\alpha \to \alpha$ in die Typumgebung eingetragen, dann muss aber

- beim Aufruf $f$ *true* die Gleichung $\alpha =$ ***bool***
- beim Aufruf $f\,1$ die Gleichung $\alpha =$ ***int***

gelöst werden. $\rightsquigarrow$ unlösbare Gleichung ***int*** $=$ ***bool***

INTUITION: Bisher können wir Typvariablen $\alpha$ nur als Platzhalter für noch unbekannte Typen benutzen und nicht als „Typparameter“.

Um Polymorphie zu bekommen, muss das Typsystem verallgemeinert werden (insbesondere müssen Typregeln „gelockert“ werden).

## 11.2 Die polymorphe Sprache $\mathcal{L}_2^{ML}$ (bzw. $\mathcal{L}_3^{ML}$)

Das Typsystem wird so erweitert, dass Typvariablen in zwei „Rollen“ auftreten können, nämlich

- als Platzhalter (wie bisher)
- als Typparameter

**Definition 27:**

1. Die Menge *PType* aller polymorphen Typen $\pi$ (auch Typschemata genannt) ist definiert durch $\pi ::= \forall\alpha_1, \ldots, \alpha_n.\tau (n \geq 1) \mid \tau$.
2. Die (bisherigen) Typen $\tau$ bezeinet man jetzt als monomorphe Typen.
3. Eine Typumgebung $\Gamma$ ist nunmehr definiert als eine endliche partielle Funktion $\Gamma : Id \hookrightarrow PType$.

INTUITION: Im polymorphen Typ $\forall\alpha_1, \ldots, \alpha_n.\tau$ dienen die quantifizierten $\alpha_i$ als Typparameter und die übrigen Typvariablen nur als Platzhalter (wie bisher). Die Typparameter dürfen – im Unterschied zu gewöhnlichen Platzhaltern – verschiedene Typen im gleichen Ausdruck annehmen.

ABSICHT: Eine mit **let** deklarierte Funktion kann einen polymorphen Typ annehmen, z. B. wird durch **let** $f = \boldsymbol{\lambda} x.x \ldots$ der Typ $\forall\alpha.\alpha \to \alpha$ für $f$ eingetragen. Der Quantor „$\forall\alpha$“ macht $\alpha$ zu einem Typparameter, d. h. $f$ darf an unterschiedlichen Stellen des Programms unterschiedliche Typen annehmen, z. B. ***int*** $\to$ ***int*** und ***bool*** $\to$ ***bool***

**Definition 28:** Ein Typurteil ist (wie vorher) von der Form $\Gamma \rhd e :: \tau$, wobei jetzt die neuartigen Typumgebungen $\Gamma$ verwendet werden.

Gültige Typurteile werden wieder durch Regeln definiert, insbesondere eine „polymorphe **let**-Regel“ (P-LET), die uns erlaubt, polymorphe Typen für Namen einzutragen, und eine „polymorphe *id*-Regel“ (P-ID), die uns erlaubt, die Polymorphie eines Namens auszunutzen (und eine entsprechende Regel für Konstanten wie [ ], *hd*, *tl*).

**Definition 29:**

- Die Menge $free(\pi)$ der in $\pi$ frei vorkommenden Typvariablen ist definiert durch: $free(\forall\alpha_1, \ldots, \alpha_n\tau) = var(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}$ mit
  (Intuition: „$\forall\alpha$“ ist ein Bindungsmechanismus wie „$\boldsymbol{\lambda} id$“ oder „**rec** $id$“)
- Die Menge $free(\Gamma)$ aller in $\Gamma$ frei vorkommenden Typvariablen ist definiert durch: $free(\Gamma) = \bigcup_{id \in dom(\Gamma)} free(\Gamma(id))$ mit $\Gamma(id) \in PType$
- Der polymorphe Abschluss eines Types $\tau$ in der Typumgebung $\Gamma$ ist definiert durch: $closure_\Gamma(\tau) = \forall\alpha_1, \ldots, \alpha_n.\tau$, wobei $\{\alpha_1, \ldots, \alpha_n\} = var(\tau) \setminus free(\Gamma)$

Die Typregeln für $\mathcal{L}_2^{ML}$ (bzw. $\mathcal{L}_3^{ML}$) entstehen aus denen für $\mathcal{L}_2^{ti}$, indem man (LET), (ID) und (CONST) durch die folgenden „polymorphen“ Regeln ersetzt:

(P-LET) $$\frac{\Gamma \rhd e_1 :: \tau_1 \quad \Gamma[closure_\Gamma(\tau_1)/id] \rhd e_2 :: \tau_2}{\Gamma \rhd \textbf{let } id = e_1 \textbf{ in } e_2 :: \tau_2}$$

(P-ID) $\Gamma \rhd id :: \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ [14] falls $id \in dom(\Gamma)$ und $\Gamma(id) = \forall\alpha_1, \ldots, \alpha_n.\tau$

(P-CONST) $\Gamma \rhd c :: \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ falls $c :: \forall\alpha_1, \ldots, \alpha_n.\tau$

Um (P-CONST) anwenden zu können, müssen wir für Konstanten polymorphe Typen $\pi$ vorgeben, nämlich:

- [ ] **::** $\forall\alpha.\alpha$ **list**
- *hd* **::** $\forall\alpha.\alpha$ **list** $\to \alpha$
- *tl* **::** $\forall\alpha.\alpha$ **list** $\to \alpha$ **list**

[14] $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ ist eine Instanz des polymorphen Typs $\forall\alpha_1, \ldots, \alpha_n.\tau$.

- *cons* :: $\forall\alpha.\alpha \to \alpha\,\mathbf{list} \to \alpha\,\mathbf{list}$
- alle anderen Konstanten wie bisher

**Beispiel 47:** *(einer Herleitung)*

(1) **let** $f = \boldsymbol{\lambda} x.x$ **in if** *true* **then** $f\,1$ **else** $f\,2$ :: ***int***
- (P-Let)
  - (2) $\boldsymbol{\lambda} x.x :: \alpha \to \alpha$
    - (Abstr)
      - (3) $[x : \alpha] \triangleright x :: \alpha$ wegen (P-Id) (da kein $\forall$-Quantor)
  - (4) $[f : \forall\alpha.\alpha \to \alpha] \triangleright$ **if** $f$ *true*$\ldots$ :: ***int*** (denn $\alpha \notin$ *free*([ ]))
    - (Cond)
      - (5) $[f : \forall\alpha.\alpha \to \alpha] \triangleright f$ *true* :: ***bool***
        - (App)
          - (6) $[f : \forall\alpha.\alpha \to \alpha] \triangleright f$ :: ***bool*** $\to$ ***bool*** wegen (P-Id)
          - (7) $[f : \forall\alpha.\alpha \to \alpha] \triangleright$ *true* :: ***bool*** wegen (P-Const)
      - (8) $[f : \forall\alpha.\alpha \to \alpha] \triangleright f\,1$ :: ***int*** analog
      - (9) $[f : \forall\alpha.\alpha \to \alpha] \triangleright f\,2$ :: ***int*** analog

**Beispiel 48:** *(zur Illustration, dass* $\text{closure}_\Gamma(\tau)$ *nicht einfacher definiert werden kann)*

(1) **let** $f = \boldsymbol{\lambda} x.$**let** $g = \boldsymbol{\lambda} y.x$ **in** $g\,0$ **in not** $(f\,1)$
- (P-Let)
  - (2) $\boldsymbol{\lambda} x.$**let** $g = \boldsymbol{\lambda} y.x$ **in** $g\,0 :: \alpha_1 \to \alpha_1$
    - (Abstr)
      - (3) $[x : \alpha_1] \triangleright$ **let** $g = \boldsymbol{\lambda} y.x$ **in** $g\,0 :: \alpha_1$
        - (P-Let)
          - (4) $\underbrace{[x : \alpha_1]}_{\Gamma} \triangleright \boldsymbol{\lambda} y.x :: \underbrace{\alpha_2 \to \alpha_1}_{\tau}$
            - (5) $[x : \alpha_1, y : \alpha_2] \triangleright x :: \alpha_1$ (P-Id)
          - (6) $[x : \alpha_1, g : \underbrace{\forall\alpha_2.\alpha_2 \to \alpha_1}_{closure_\Gamma(\tau)}] \triangleright g\ 0 :: \alpha_1$
            - (App)
              - (7) $[x : \alpha_1, g : \forall\alpha_2.\alpha_2 \to \alpha_1] \triangleright g$ :: ***int*** $\to \alpha_1$ (P-Id)
              - (8) $[x : \alpha_1, g : \forall\alpha_2.\alpha_2 \to \alpha_1] \triangleright 0$ :: ***int*** (P-Const)
  - (9) $[f : \underbrace{\forall\alpha_1.\alpha_1 \to \alpha_1}_{closure_{[\,]}(\alpha_1 \to \alpha_1)}]$**not**$(f\ 1)$ :: $\lightning$
    - (App)
      - (10) $[f : \forall\alpha_1.\alpha_1 \to \alpha_1]$**not** :: ***bool*** $\to$ ***bool***
      - (11) $[f : \forall\alpha_1.\alpha_1 \to \alpha_1] f\ 1$ :: ***int***
        - (App)
          - (12) $[f : \forall\alpha_1.\alpha_1 \to \alpha_1] f$ :: ***int*** $\to$ ***int*** (P-Id)
          - (13) $[f : \forall\alpha_1.\alpha_1 \to \alpha_1] 1$ :: ***int*** (P-Const)

Beachte: $f$ ist die Identität, da $g\,0$ den Parameter $x$ von $f$ zurückliefert. Also führt **not** $(f\,1)$ zum Laufzeitfehler.

Wenn es in der Regel (P-Let) erlaubt wäre, über alle Typvariablen von $\tau$ zu quantifizieren, dann wäre unser Ausdruck wohlgetypt ($\rightsquigarrow$ ***bool***). Damit wäre die Typsicherheit verletzt.

## 11.3 Typsicherheit von $\mathcal{L}_2^{ML}$

BISHER: Typsicherheit für $\mathcal{L}_2^t$ und $\mathcal{L}_2^{ti}$ ließ sich auf die Typsicherheit von $\mathcal{L}_2$ zurückführen, da aus einer Herleitung für $e :: \tau$ in $\mathcal{L}_2^t$ bzw. $\mathcal{L}_2^{ti}$ leicht eine Herleitung für $\mathit{erase}(e) :: \tau$ in $\mathcal{L}_2$ entsteht (und weil die Semantik von $e$ als Semantik von $\mathit{erase}(e)$ definiert ist).

JETZT: Wieder wird die Semantik von $e$ als die von $\mathit{erase}(e)$ definiert.
Aber: Durch die Polymorphie ist es möglich, dass $e$ in $\mathcal{L}_2^{ML}$ wohlgetypt ist, aber $\mathit{erase}(e)$ nicht wohlgetypt in $\mathcal{L}_2$.

**Beispiel 49:** **let** $f = \boldsymbol{\lambda} x.x$ **in if** $f\ \mathit{true}$ **then** $f\ 1$**else** $f\ 2$
ist wohlgetypt in $\mathcal{L}_2^{ML}$, aber nicht in $\mathcal{L}_2$ oder $\mathcal{L}_2^{ti}$.

Also kann man nicht analog zu $\mathcal{L}_2^t$ oder $\mathcal{L}_2^{ti}$ argumentieren.
$\rightsquigarrow$ Die Typsicherheit muss bewiesen werden.

IDEE ZUM BEWEIS: (wie vorher) Preservation+Progress.
Der Beweis von Progress bleibt im Wesentlichen der Gleiche.
Beim Beweis von Preservation spielen die neuen Typregeln (insb. (P-LET)) eine Rolle. Zur Vorbereitung des Beweises folgen nun einige Lemata.

**Definition 30:** *(Anwendung einer Substitution auf polymorphe Typen und Typumgebungen)*

1. Sei $\pi = \forall \alpha_1, \ldots, \alpha_n : \tau$ und $s \in \mathit{Subst}$.
   Dann ist $\pi\, s = \forall \alpha_1, \ldots, \alpha_n : \tau\, \left(s\,|_{var(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}}\right)$
   Man beachte: In $\tau$ werden nur die freien Typvariablen $\alpha$ durch $s\,(\alpha)$ ersetzt, also alle $\alpha \neq \alpha_1, \ldots, \alpha_n \Rightarrow$ „$\forall$“ ist ein Bindungsmechanismus.
2. $\Gamma\, s$ ist die Typumgebung, die aus $\Gamma$ entsteht, indem man jeden Eintrag $id : \pi$ durch $id : \pi\, s$ ersetzt. (d. h.: $(\Gamma\, s)(id) = \Gamma(id)\, s$)

**Definition 31:** *(Instanz eines polymorphen Typen)*
Sei $\pi = \forall \alpha_1, \ldots, \alpha_n : \tau$.
Dann nennt man Typen der Form $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ Instanzen von $\pi$.

**Lemma 15:** *(Spezialisierung von Typurteilen)*
*Wenn $\Gamma \rhd e :: \tau$ gültig ist, dann ist auch $\Gamma s \rhd e :: \tau s$ gültig.*

BEWEISSKIZZE: In der Herleitung von $\Gamma \rhd e :: \tau$ ersetzt man jedes freie Vorkommen einer Typvariablen $\alpha$ durch $s\,(\alpha)$. $\rightsquigarrow$ Es entsteht eine Herleitung für $\Gamma s \rhd e\, s :: \tau s$.
(exakter Beweis: Induktion über die Länge der Herleitung)

**Korollar 3:** *Wenn $\Gamma \rhd e :: \tau$ gültig ist und $\alpha_1, \ldots, \alpha_n \notin \mathit{free}(\Gamma)$,*
*dann gilt auch $\Gamma \rhd e[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] :: \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ für alle $\tau_1, \ldots, \tau_n \in \mathit{Type}$.*

**Beweis:** Der Beweis folgt aus dem Lemma weil $\Gamma[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] = \Gamma$. □

ALTERNATIVE FORMULIERUNG: Wenn $\Gamma \rhd e :: \tau$ gültig ist und $\tau'$ ist eine Instanz des polymorphen Typs $\mathit{closure}_\Gamma(\tau)$, dann gilt $\Gamma \rhd \mathit{erase}(e) :: \tau'$.

**Beweis:** $\mathit{closure}_\Gamma(\tau) = \forall \alpha_1, \ldots, \alpha_n : \tau$, wobei $\alpha_1, \ldots, \alpha_n$ alle Typvariablen in $\tau$ sind, die nicht frei in $\Gamma$ vorkommen. Also ist $\tau'$ von der Form $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ mit $\alpha_1, \ldots, \alpha_n \notin \mathit{free}(\Gamma)$.
$\underset{\text{Korollar}}{\Rightarrow}$ $\Gamma \rhd e[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] :: \tau'$, also auch $\Gamma \rhd \mathit{erase}(e) :: \tau'$. □

**Lemma 16:** *(Typerhaltung bei Substitution)*
*Wenn $\Gamma \rhd e :: \tau$ und $\Gamma[\mathit{closure}_\Gamma(\tau)/id] \rhd e' :: \tau'$, dann gilt $\Gamma \rhd e'[\mathit{erase}(e)/id] :: \tau'$.*
*(Vergleich zu früher: Statt $\Gamma[\tau/id]$ wird jetzt $\Gamma[\mathit{closure}_\Gamma(\tau)/id]$ benutzt.)*

BEWEISSKIZZE: Wir betrachten eine Herleitung von $\Gamma[closure_\Gamma(\tau)/id] \rhd e' :: \tau'$.
In dieser Herleitung muss man die Regel (P-ID) für die freien Vorkommen von $id$ in $e'$ benutzen.
$\rightsquigarrow$ Man wählt für jedes solche Vorkommen von $id$ eine Instanz $\tau''$ des polymorphen Typs $closure_\Gamma(\tau)$, wobei man an verschiedenen Stellen verschiedene Instanzen wählen kann.
Nach dem Korollar lässt sich für jede solche Instanz $\tau''$ das Typurteil $\Gamma \rhd erase(e) :: \tau''$ herleiten. Also können wir $erase(e)$ für jedes freie Vorkommen von $id$ einsetzen und anstelle von (P-ID) die Herleitung $\Gamma \rhd erase(e) :: \tau'$ einfügen.
Damit entsteht eine Herleitung für $\Gamma \rhd e'[erase(e)/id]$.

**Satz 13:** *(Preservation für $\mathcal{L}_2^{ML}$)*
*Wenn $e \to e'$ und $\Gamma \rhd e :: \tau$ (mit den Typregeln von $\mathcal{L}_2^{ML}$), dann gilt auch $\Gamma \rhd erase(e') :: \tau$.*
*(Insbesondere gilt $\Gamma \rhd e' :: \tau$, wenn $e$ keine Typvariablen enthält.)*

**Beweis:** Interessant ist nur der small step für (LET-EXEC), weil nur dort die neue Regel (P-LET) ins Spiel kommt.
Wenn **let** $id = v$ **in** $e \to e[v/id]$ mit (LET-EXEC) und $\Gamma \rhd$ **let** $id = v$ **in** $e :: \tau$, dann kann letzterer nur mit (P-LET) entstanden sein aus
(1) $\Gamma \rhd v :: \tau'$
(1) $\Gamma[closure_\Gamma(\tau')/id] \rhd e :: \tau$
Also gilt nach dem Lemma über Substitution $\Gamma \rhd e[erase(v)/id] :: \tau$,
also auch $\Gamma \rhd erase(e[v/id]) :: \tau$. □

**Satz 14:** *(Progress)*
*Sei $e :: \tau$ (in $\mathcal{L}_2^{ML}$).*
*Dann gilt wieder $e \in Val$ oder es existiert ein small step $e \to e'$ oder $e \to \uparrow exn$.*
*Beweis wie früher.*

**Satz 15:** *(Typsicherheit)*
*Wenn $e :: \tau$ (in $\mathcal{L}_2^{ML}$), dann bleibt die Berechnung für $e$ nicht stecken.*

**Beweis:** Da die Berechnung für $e$ als die Berechnung für $erase(e)$ definiert ist, folgt der Satz wie früher aus Preservation und Progress. □

## 11.4 Typinferenz für $\mathcal{L}_2^{ML}$

PRINZIP: (wie früher) Die Funktion $solve()$ wird auf einer Menge von Gleichungen und Typurteilen angewandt und liefert eine Lösung für die Formelmenge (oder „nicht lösbar“).

**Definition 32:** Sei $\pi = \forall\alpha_1, \ldots, \alpha_n.\tau \in PType$.
Eine neue allgemeinste Instanz von $\pi$ ist ein polymorpher Typ der Form $\tau[\alpha'_1/\alpha_1, \ldots, \alpha'_n/\alpha_n]$, wobei $\alpha'_1, \ldots, \alpha'_n$ neue, paarweise verschiedene Typvariablen sind.
(Beachte: $n = 0 \rightsquigarrow \tau$ einzige (neue allgemeinste) Instanz von $\tau$)

Der Typinferenzalgorithmus für $\mathcal{L}_2^{ML}$ ergibt sich aus dem für $\mathcal{L}_2$, indem man (CONST), (ID) und (LET) durch die folgenden Regeln ersetzt:

(P-CONST) $solve(\{\Gamma \rhd c :: \tau\} \cup F) = solve(\{\tau = \tau'\} \cup F)$
wobei $c :: \pi$ und $\tau'$ neue allgemeinste Instanz von $\pi$

(P-ID) $solve(\{\Gamma \rhd id :: \tau\} \cup F) = solve(\{\tau = \tau'\} \cup F)$
falls $id \in dom(\Gamma)$, $\Gamma(id) = \pi$ und $\tau'$ neue allgemeinste Instanz von $\pi'$
„nicht lösbar“, falls $id \notin dom(\Gamma)$

(P-LET) $solve(\{\Gamma \rhd_1 \in e_2 :: \tau\} \cup F) = s_1 s_2$
falls $s_1 = solve(\Gamma \rhd e_1 :: \alpha)$ mit neuer Typvariablen $\alpha$
und $s_2 = solve(\Gamma s_1[closure_{\Gamma s_1}(\alpha s_1)/id] \rhd e_2 s_1 :: \tau s_1 \cup F s_1)$ mit neuer Typvariablen $\alpha$

BEACHTE: Anwendung von (P-LET) erfordert als Erstes die Lösung des Typurteils $\Gamma \rhd e_1 :: \alpha$. Nur mit dieser Lösung $s_1$ kann man weiterarbeiten.
Insbesondere: Im Gegensatz zum alten Algorithmus ist es hier nicht möglich, die Lösungen von Typgleichungen ans Ende zu verschieben.

**Beispiel 50:** $solve(\{\textbf{let } f = \boldsymbol{\lambda} x.x \textbf{ in if } f\ true \textbf{ then } f\,1 \textbf{ else } f\,2\} :: \alpha) \underset{\text{(P-LET)}}{=} s_1\, s_2$ wobei

$s_1 = solve(\{\boldsymbol{\lambda} x.x :: \alpha_1\})$
$\underset{\text{(ABSTR)}}{=} solve(\{[x :: \alpha_2] \rhd x :: \alpha_3, \alpha_1 = \alpha_2 \to \alpha_3\})$
$\underset{\text{(P-ID)}}{=} solve(\{\alpha_2 = \alpha_3, \alpha_1 = \alpha_2 \to \alpha_3\})$
$\underset{\text{(2x UNIFY+EMPTY)}}{=} [\alpha_2/\alpha_3, \alpha_2 \to \alpha_2/\alpha_1]$

$s_2 = solve(\{[f : \underbrace{closure_{[]}(\underbrace{\alpha_1\, s_1}_{\alpha_2 \to \alpha_2})}_{\forall \alpha_2.\alpha_2 \to \alpha_2}] \rhd \textbf{if } f\ true \textbf{ then } f\,1 \textbf{else} f\,2 :: \alpha s_1\})$

$\underset{\text{(COND)}}{=} solve(\{[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd f\ true :: \mathbf{bool},$
$[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd f\,1 :: \alpha,$
$[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd f\,2 :: \alpha\})$
$\underset{\text{(3x APP)}}{=} solve(\{[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd f :: \alpha_4 \to \mathbf{bool},$
$[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd true :: \alpha_4,$
$[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd f :: \alpha_5 \to \alpha,$
$[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd 1 :: \alpha_5,$
$[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd f :: \alpha_6 \to \alpha,$
$[f.\forall \alpha_2.\alpha_2 \to \alpha_2] \rhd 2 :: \alpha_6\})$
$\underset{\text{(3x P-ID = 3x CONST)}}{=} solve(\{\alpha_7 \to \alpha_7 = \alpha_4 \to \mathbf{bool}, \alpha_4 = \mathbf{bool},$
$\alpha_8 \to \alpha_8 = \alpha_5 \to \alpha, \alpha_5 = \mathbf{int},$
$\alpha_9 \to \alpha_9 = \alpha_6 \to \alpha, \alpha_6 = \mathbf{int}\})$

$= [\mathbf{int}/\alpha, \mathbf{bool}/\alpha_4, \mathbf{int}/\alpha_5, \mathbf{int}/\alpha_6, \mathbf{bool}/\alpha_7, \mathbf{int}/\alpha_8, \mathbf{int}/\alpha_9]$
Insgesamt: $s_1\, s_2 = [\mathbf{int}/\alpha, \alpha_2 \to \alpha_2/\alpha_1, \ldots]$

BEACHTE: Es war wichtig, dass man stets neue Instanzen des polymorphen Typs $\forall \alpha_2.\alpha_2 \to \alpha_2$ wählt, nämlich $\alpha_4 \to \alpha_4, \alpha_5 \to \alpha_5, \alpha_6 \to \alpha_6$.
$\leadsto$ Es war möglich $\alpha_4 = \mathbf{bool}$ und $\alpha_5 = \alpha_6 = \mathbf{int}$ zu wählen.

# 12 Übersicht über die eingeführten Sprachen

| | reiner $\boldsymbol{\lambda}$-Kalkül | + Basistypen,<br>+ **if-then-else** | + Rekursion | + Listen,<br>+ Tupel |
|---|---|---|---|---|
| ungetypt | $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ |
| explizit getypt | $\mathcal{L}_0^t$ | $\mathcal{L}_1^t$ | $\mathcal{L}_2^t$ | |
| implizit getypt<br>(mit Typinferenz) | $\mathcal{L}_0^{ti}$ | $\mathcal{L}_1^{ti}$ | $\mathcal{L}_2^{ti}$ | ($\mathcal{L}_3^{ti}$) mit polymorphen Konstanten [], *hd*, *tl* |
| **let**-Polymorphie | $\mathcal{L}_0^{ML}$ | $\mathcal{L}_1^{ML}$ | $\mathcal{L}_2^{ML}$ | $\mathcal{L}_3^{ML}$ |

Ab der zweiten Zeile gilt Typsicherheit, d. h. abgeschlossene, wohlgetypte Programme bleiben nicht stecken.
In $\mathcal{L}_1$, $\mathcal{L}_2$ und $\mathcal{L}_3$ können (auch abgeschlossene) Ausdrücke stecken bleiben.
In $\mathcal{L}_0$ bleiben nur Ausdrücke mit freien Namen stecken (nicht bewiesen).
Für alle hinterlegten Sprachen gilt: Alle Programme terminieren, da keine Rekursion möglich ist.
In $\mathcal{L}_0$ und $\mathcal{L}_1$ ist Divergenz möglich, obwohl keine explizite Rekursion vorhanden ist, z. B. bei dem Ausdruck $(\boldsymbol{\lambda} x.x\,x)(\boldsymbol{\lambda} x.x\,x)$.

BEACHTE: $\boldsymbol{\lambda} x.x\,x$ ist selbst mit **let**-Polymorphie nicht wohlgetypt, denn der Name $x$ wird mit $\boldsymbol{\lambda}$ eingeführt, d. h. es wird kein polymorpher Typ für $x$ eingetragen.

# 13 Ausblick

TP II: Erweiterung von $\mathcal{L}_2$ durch imperative Konzepte (Zuweisung, Pointer, **while**, **for**, Sequenz), zugehöriges Typsystem, auch mit **let**-Polymorphie, Typsicherheit, Typinferenz, Objektorientierung $\rightsquigarrow$ Subtyping, Vererbung, rekursive Typen

TP III: Denotationelle Semantik