

Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

11. Dezember 2007

Ziele des Kapitels:

- Polymorphes Typsystem für alle Konstrukte von KFPTSP
- Typsystem unabhängig vom Haskell-Typcheck
Soll das Typsystem von Haskell verständlich machen
- Funktionalität des Formalismus
- Grenzen; und Vor- und Nachteile
- zwei Varianten des Typchecks

Urväter dieses Typsystems:

Hindley, **Milner**, Mycroft, Abramsky, Damas

Annahmen

KFPTSP ist die Grundlage:

- Es gibt eine Typsyntax, Typen und Typkonstruktoren
- Der Typ der Daten-Konstrukturen ist vorgegeben
- Ein case-Konstrukt pro abstraktem Datentyp
(mit `data type` deklariert)
(nicht für den Funktionsdatentyp $a \rightarrow b$)
- rekursive Superkombinatoren sind erlaubt.

- Wenn Typcheck erfolgreich,
dann keine dynamischen Typfehler (d.h. zur Laufzeit)
- Flexibles Programmieren trotz Typenbeschränkungen
- Typisierung sollte entscheidbar sein.
Falls möglich: effizient
- Jeder Ausdruck und Unterausdruck muss einen Typ haben.

Exp ::= V (Variablen)
| $(\backslash V \rightarrow Exp)$
| $(Exp_1 Exp_2)$
| $(c_{Typ} Exp_1 \dots Exp_n)$ (wobei $n = ar(c)$)
| $(case_{Typ} Exp \text{ of } \{Pat_1 \rightarrow Exp_1; \dots; Pat_n \rightarrow Exp_n\})$
 (Pat_i sind alle einfachen Pattern zum Typ des case)

Pat ::= $(c V_1 \dots V_{ar(c)})$
 Die Variablen V_i müssen alle verschieden sein.

Definition: Ein dynamischer Typfehler tritt auf, wenn:

Die Normalordnung versucht folgendes auszuwerten:

- `(case_Typ (c ...))` wobei c kein Konstruktor zu `Typ` ist.
- `(case_Typ (\x->s))`
- `(case_Typ (sc t1 .. tn))` und $n < ar(sc)$
- `((c ...) t)` wobei c ein Konstruktor ist.

Satz Es gibt **kein (entscheidbares) Typsystem** in KFPT (KFPTS),
das genau die Ausdrücke erkennt,
die keine dynamischen Typfehler verursachen.

Beweis-Skizze Zurückführung auf das Halteproblem:

Betrachte den Ausdruck

```
if t then case_Bool Nil {p1-> (Nil Nil) ; ... }  
      else case_Bool Nil {p1-> (Nil Nil) ; ... }
```

Falls t nur True, False, (oder Nichtterminierung) ergeben kann,
dann ergibt sich ein Typfehler genau dann wenn t terminiert.

Als t kann man eine Turingmaschine in KFPT kodieren

D.h.: Lösung des Halteproblem \Leftrightarrow Existenz eines exakten Typcheckers

- Angenommen, wir haben ein entscheidbares Typsystem, das keine dynamischen Typfehler zulässt.
Dann gibt es ungetypte Ausdrücke, die keine dynamischen Typfehler verursachen.
- Erweiterung des entscheidbaren Typsystems sind immer möglich;
sind trotzdem **nie exakt**.

Syntax

Typ ::= Typkonstante

| Typvariable

| $(TC \ Typ_1 \dots \ Typ_n)$ Wobei TC Stelligkeit n hat.

elementarer Typ

Grundtyp, monomorpher Typ.

polymorph

= Basistyp, d.h. Konstante

= Typ ohne Typvariablen

= mit Typvariablen

(Typ-)Variable: Gleiche Variablen bedeuten gleichen Typ.

Typkonstruktor: zu jedem algebraischen Datentyp gibt es einen.
Die Stelligkeit der Konstruktoren ist festgelegt.

→ zweistelliger Typ-Konstruktor für Funktionen.

Zwei Möglichkeiten:

- (A) Als Menge von Grundtypen
- (B) Als Formel erster Ordnung:
Aussage über Argument und Ergebnistypen einer Funktion

Beispiel: `Cons: $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$`

- (A) Die Menge der Typen von `Cons` ist:
 $\{\tau \rightarrow [\tau] \rightarrow [\tau] \mid \tau \text{ ist ein Grundtyp}\}$.
- (B) $\forall \alpha. \alpha \rightarrow [\alpha] \rightarrow [\alpha]$:
Für alle Eingaben ist die Ausgabe vom Typ $[\alpha]$.

Betrachtungsweise: Jeder Ausdruck hat eine Menge von Grundtypen

Annahmen:

- Typ des Datenkonstruktors gibt die **Stelligkeit** an, und die möglichen Typen und Typkombinationen der Argumente.
- **Zieltyp** muss von der Form $T \alpha_1 \dots \alpha_n$ sein, wobei T Typkonstruktor des zugehörigen algebraischen Datentyps und die α_i verschiedene Typvariablen sind.
- Nur $\alpha_1 \dots \alpha_n$ können als Typvariablen in Argumenttypen von Konstruktoren vorkommen

Das schließt im Moment GADT aus, ebenso existenzielle Typen in Konstruktor-Deklarationen

Paare: $(.,.) : \alpha \rightarrow \beta \rightarrow (\alpha, \beta)$

Listen: Nil: $[\alpha]$ und ':' : $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$

Boolesche Werte: True: Bool und False: Bool.

```
Data Tree a = Node a (Tree a) (Tree a) | Leaf a
```

- Tree ist der einstellige Typkonstruktor.
- Node, Leaf sind die (Daten-)Konstruktoren.
- Node: $\alpha \rightarrow (\text{Tree } \alpha) \rightarrow (\text{Tree } \alpha) \rightarrow (\text{Tree } \alpha)$
- Leaf: $\alpha \rightarrow (\text{Tree } \alpha)$

(Typ-)Substitution σ : ist Abbildung : Typ-Ausdrücke \mapsto Typen
definiert auf den Typ-Variablen

Anwendungsregeln: $\sigma^* \alpha = \sigma \alpha$
 $\sigma^*(c \ t_1 \dots t_n) = (c \ (\sigma^* t_1) \dots (\sigma^* t_n))$

Beispiel

Instanzen von $[\alpha]$ sind $[\text{Int}]$ mit $\{\alpha \mapsto \text{Int}\}$

$[[\alpha]]$ mit $\{\alpha \mapsto [\alpha]\}$

$[\beta \rightarrow \text{Int}]$ mit $\{\alpha \mapsto (\beta \rightarrow \text{Int})\}$

Idee der Typregeln:

- Die Regeln berechnen **Grundtypen** von Ausdrücken
- – (i.a.unendlich viele) Grundtypen –

Ist scheinbar umständlich,
Aber allgemeiner als polymorphe Typ-Schemata.

Aufbau einer Regel:

bisherige Folgerungen
neue Folgerung

Typannahme $\vdash t : \tau$ ist eine Einzel-Komponente (judgement)

“Bisherige Folgerungen” = Liste von Komponenten

“neue Folgerungen” bestehen aus einer Komponente

Regeln ohne Voraussetzungen sind **Axiome**.

Bei diesen wird die Annahme und der Strich weggelassen.

- Typannahmen A :** sind eine Menge von Annahmen über die möglichen Typen der freien Variablen und bekanntes Wissen über die Typen der Konstanten bzw. Funktionssymbole
- Variablen x** haben höchstens eine Typannahme $x : \tau$ in jedem A
Superkombinatoren werden getrennt behandelt
- Konstanten c** können mehrere Typannahmen haben: $c : \tau_1, \dots, c : \tau_n$.
I.A. Menge aller Grundinstanzen eines polymorphen Typs
Typangabe $c : \forall(\sigma)$.

Die Typregeln definieren eine **Herleitbarkeitsrelation** \vdash

$A \vdash s : \tau$ bedeutet, dass aus A der Typ τ für s herleitbar ist.

I.a. ist **eine Menge von Typen** für einen Term herleitbar.

Oft ist diese Menge eine Instanzmenge eines polymorphen Typs.

Mit der Typmengen-Methode kann man einige Beschränkungen des Typklassensystem modellieren:

Typen von \vdash :

$\{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}, \dots\}$.

Das sind endlich viele;

Diese Menge ist **nicht** als Instanz eines polymorphen Typs darstellbar.

τ, τ_i sind Grundtypen, T Menge von Grundtypen.

Axiom für Konstanten

$$A \cup \{c : T\} \vdash c : \tau \text{ wenn } \tau \in T$$

Axiom für Konstanten (polymorphe Typen)

$$A \cup \{c : \forall(\sigma)\} \vdash c : \tau' \text{ wenn } \tau' \text{ Grundinstanz von } \sigma$$

Axiome für freie Variablen

$$A \cup \{x : \tau\} \vdash x : \tau$$

Anwendungsregel:

$$\frac{A \vdash a : (\tau_1 \rightarrow \tau_2); A \vdash b : \tau_1}{A \vdash (a \ b) : \tau_2}$$

Superkombinator-Regel

$$\frac{A \cup \{x_1 : \tau_1, \dots, x_n : \tau_n\} \vdash e : \tau; \text{sc } x_1 \dots x_n = e \text{ ist die Definition für den Superkombinator } \text{sc}}{A \vdash \text{sc} : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau}$$

Nebenbedingung: A hat keine Annahmen über x_i

Abstraktion:

$$\frac{A \cup \{x : \tau_1\} \vdash t : \tau_2}{A \vdash (\lambda x.t) : \tau_1 \rightarrow \tau_2}$$

case-Regel:

μ ist Typ zu Typkonstruktor D

$A \vdash e : \mu$

$A \cup \{x_{1,1} : \tau_{1,1} \dots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash (c_1 x_{1,1} \dots x_{1,n(1)}) : \mu$

...

$A \cup \{x_{m,1} : \tau_{m,1} \dots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash (c_m x_{m,1} \dots x_{m,n(m)}) : \mu$

$A \cup \{x_{1,1} : \tau_{1,1} \dots x_{1,n(1)} : \tau_{1,n(1)}\} \vdash e_1 : \tau$

...

$A \cup \{x_{m,1} : \tau_{m,1} \dots x_{m,n(m)} : \tau_{m,n(m)}\} \vdash e_m : \tau$

$A \vdash (\text{case}_D e \{(c_1 x_{1,1} \dots x_{1,n(1)}) \rightarrow e_1; \dots; (c_m x_{m,1} \dots x_{m,n(m)}) \rightarrow e_m\}) : \tau$

Es reicht aus, sich zu merken:

- e und alle Pattern haben gleichen Typ der zu D gehört ebenso die entsprechenden Unterterme
- alle e_i und der Gesamt-case-Ausdruck haben gleichen Typ.

Die Typregeln für Abstraktionen, Superkombinatoren und case verändern die Menge der Typannahmen:

und zwar um lokale Annahmen für gebundene Variablen

Obige Regeln sind ausreichend für Typen von **nicht** rekursiv definierten Superkombinatoren

Man stellt fest, dass oft polymorphe Typen (Schemata) für die Menge der herleitbaren Typen eines (nicht-rekursiven) Superkombinatoren herleitbar sind.

Definition

Ein Ausdruck t ist (bzgl des Grundtypensystem) *wohlgetypt*, wenn man für t mindestens einen Typ herleiten kann.

D.h. $A \vdash t :: \tau$ soll herleitbar sein.

Hierbei darf A nur Annahmen zu Variablen enthalten, die frei in t vorkommen.

Es gilt

Bei geschlossenen Ausdrücken muss $A = \emptyset$ sein

Wenn t wohlgetypt ist, dann auch alle Unterausdrücke von t .

Abstraktion $Id := \lambda x.x$:

$$\frac{\{x : \tau\} \vdash x : \tau}{\emptyset \vdash \lambda x.x : \tau \rightarrow \tau}$$

wobei τ ein (beliebiger) Grundtyp ist.

D.h. $\lambda x.x$ hat alle Typen der Form $\{\tau \rightarrow \tau\}$.

Das ist gerade die Menge der Instanzen von $\alpha \rightarrow \alpha$.

`unitList x = Cons x Nil`

Menge der Annahmen: $A = \{x : \tau, (\text{Nil} : \forall[\beta]), \text{Cons} : \forall(\alpha \rightarrow [\alpha] \rightarrow [\alpha])\}$

1. $A \vdash (\text{Cons } x) : [\tau] \rightarrow [\tau]$
2. $A \vdash (\text{Cons } x \text{ Nil}) : [\tau]$
3. Superkombinatorregel: `unitList`: $\tau \rightarrow [\tau]$.

Für alle Grundtypen τ gilt: `unitList`: $\tau \rightarrow [\tau]$.

Auch hier: `unitList` : $\alpha \rightarrow [\alpha]$

`twice f x = f (f x)`

Typannahme $A = \{x : \tau_1, f : \tau_2\}$.

1. Der Typ von `(f x)` muss existieren: deshalb muss $f : \tau_2 = \tau_1 \rightarrow \tau_3$ sein, wobei τ_3 noch unbestimmt ist.
 $a \vdash (f\ x) : \tau_3$.
2. Nur Annahmen der Form $A' = \{x : \tau_1, f : \tau_1 \rightarrow \tau_3\}$ sind sinnvoll
3. Typ von `f (f x)` muss existieren:
Argumentvariablen haben nur einen Typ:
Also $\tau_1 = \tau_3$ und $f\ (f\ x) : \tau_1$.
4. $A'' = \{x : \tau_1, f : \tau_1 \rightarrow \tau_1\}$
herleitbar: `f (f x) : τ_1`
5. Mit der Superkombinatorregel: `twice`: $(\tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$.
6. Polymorpher Typ von `twice`: $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

Statt mit $x : \tau$ Grundtypen “durchzuprobieren”:
 $x : \alpha$ mit Typvariable α .

Wesentlich für das Typsystem:

Nur **eine** Grundtypannahme für lokale Variablen (ohne Quantoren)
keine Menge o.ä.

Lambda-gebundene Variablen

dürfen **nicht** polymorph verwendet werden

Das ist eine echte Einschränkung, falls Variablen vom Funktionstyp

`selfapply` kommt als Unterausdruck in der Definition von Y vor.

Typ von `selfapply x = x x` nach zwei verschiedenen Methoden.

1. lambda-gebundene Variablen sind monomorph
2. Argumente dürfen eine Menge von Typen haben.
(was wäre wenn, ...)
Polymorphismus mit höherem Rang.

`selfapply x = x x`

Unter der Annahme, dass lambda-gebundene Variablen monomorph sind.

1. Annahme: $\{x : \tau\}$
2. $(x x)$ hat Typ, deshalb $\tau = \tau_1 \rightarrow \tau_2$ und $\tau_1 = \tau$. Fehler, denn solche Grundtypen, gibt es nicht.

Aber: $(\text{selfapply } x)$ kann vernünftig sein:

Z.B: $\text{selfapply } I \rightarrow (I I) \rightarrow I$.

Damit hat $Y = \lambda f.(\lambda x.f(x x)) (\lambda x.f(x x))$ **keinen** polymorphen Typ

Als virtuelle Erweiterung des Typsystems:
eine Menge von Typen ist erlaubt pro Variable

1. $\{\text{selfapply} : S; x : T\}$
2. $(x \ x)$ muss einen Typ haben $\Rightarrow T$ muss erfüllen:
Es gibt Typen $\tau_1 \in T$ und $\tau_1 \rightarrow \tau_2 \in T$
3. Die von T abgeleitete Typmenge von $(x \ x)$ ist dann:
 $T_1 = \{\tau_2 \mid \text{Es gibt } \tau_1 \rightarrow \tau_2 \in T \text{ und } \tau_1 \in T\}$
D.h. $\text{selfapply} : T \rightarrow \{\tau_2 \mid \text{Es gibt } \tau_1 \rightarrow \tau_2 \in T \text{ und } \tau_1 \in T\}$

Der Mengen-Typ von selfapply ist
keine Instanz eines polymorphen Typausdrucks,
sondern eine komplexere Bedingung an Mengen von Grundtypen.

Das bisherige Typsystem kann keine Typen für rekursiv definierte Superkombinatoren herleiten, da die Definition der Herleitbarkeit dann zyklisch ist.

Erweiterung zur Berechnung von

Typen von rekursiven Superkombinatoren

Konsistente Annahme

- Programm: $sc_1 = e_1, \dots, sc_n = e_n$
- $A_C :=$ Typen der Konstruktoren
- $A_{SC} := \cup \{sc_1 :: S_1, \dots, sc_n :: S_n\}$
sind Typannahmen zu den sc_i
- Für jeden Kombinator sc_j und für alle $\tau_i \in S_j$:
lässt sich jeder angenommene Typ herleiten, d.h.:
 $A_{SC} \cup A_C \vdash sc_j : \tau_i$,
wobei die letzte angewendete Regel jeder Herleitung
jeweils die Superkombinatorregel zu sc_j ist,
- dann ist A eine *konsistente Typannahme*.

A ist dann ein **Fixpunkt** des Verfahrens.

Konsistenz: alle Typen von Superkombinatoren,
die in den Annahmen sind,
kann man auch aus diesen Annahmen herleiten
(Evtl. mehr Typen)

Definition

Ein Programm P (eine Menge von Superkombinatordefinitionen) ist *wohlgetypt*,
wenn es eine konsistente Typannahme zu P gibt.

Wenn es eine allgemeinste (größte) konsistente Typannahme gibt
dann sind $sc_i : S_i$ die **allgemeinsten Typen** der Superkombinatoren.

`repeat x = Cons x (repeat x)`

Versuch einer Typannahme:

$A = \{\text{repeat} : \forall \alpha, \text{Cons} : \forall (\beta \rightarrow [\beta]) \rightarrow [\beta], x : \tau\}$

Dann ergibt sich:

1. `(repeat x)` (rechts) ist getypt, deshalb $\text{repeat} : \tau \rightarrow \alpha'$
2. `(Cons x (repeat x))` ist getypt, deshalb $[\tau] = \alpha'$
und damit $(\text{Cons x (repeat x)}) : [\tau]$.
3. Das Ergebnis: $\text{repeat} : \tau \rightarrow [\tau]$

Das geht für alle τ !

Neue Annahmen $A_1 = \{\text{repeat} : \forall(\alpha \rightarrow [\alpha]), \text{Cons} : \forall(\beta \rightarrow [\beta] \rightarrow [\beta])\}$

Konsistenz-Test bzw Typisierung unter A_1 :

1. $(\text{repeat } x) : [\tau]$
2. $(\text{Cons } x (\text{repeat } x)) : [\alpha]$ und $\beta = \alpha = \tau$
3. Damit: $\text{repeat} : \tau \rightarrow [\tau]$ für alle τ
mit Superkombinatorregel als letzte Regel herleitbar
4. Damit ist A_1 eine konsistente Typannahme.

Aussage Für direkt ungetypte Ausdrücke lässt sich kein Typ herleiten.

direkt ungetypte Fälle:

1. $(\text{case_T } (c \dots) \dots)$ wobei c kein Konstruktor zu Typ ist.
2. $(\text{case_T } (\backslash x \rightarrow s) \dots)$
3. $(\text{case_T } (sc \ t1 \dots tn) \dots)$ und $n < ar(sc)$
4. $((c \dots) \ t)$ wobei c ein Konstruktor ist.

zu 1. Keine Typregel für case anwendbar:
Pattern-Typ verschieden von $(T \dots)$

zu 2. Abstraktion hat Typ $\tau_1 \rightarrow \tau_2 \neq (T \dots)$

zu 3. $(sc \ t1 \dots tn)$ hat Typ $\tau_1 \rightarrow \tau_2 \neq (T \dots)$

zu 4. $((c \ t1 \dots tn) \ t)$: Typ von $(c \ t1 \dots tn)$ ist kein \rightarrow -Typ
 \implies Die Typregel für Anwendungen ist nicht anwendbar

Lemma Sei A konsistente Typannahme für alle Superkombinatoren.

Dann gilt:

Wenn $t \rightarrow t'$ eine Reduktion ist,

Dann $A \vdash t : \tau \implies A \vdash t' : \tau$ (falls t geschlossen)

Bzw. $A \cup A_V \vdash t : \tau \implies A \cup A_V \vdash t' : \tau$ (falls t offen)

Begründung: Alle Reduktionsfälle durchgehen:

Superkombinator-Reduktion, Beta-Reduktion, Case-Reduktion

1. Superkombinator-Reduktion:

$$sc\ t_1 \dots t_n \rightarrow e[t_1/x_1 \dots t_n/x_n] \quad \text{mit } sc\ x_1 \dots x_n = e$$

Wir wollen wissen, ob:

$$(A \vdash sc\ t_1 \dots t_n) :: \tau \implies (A \vdash e[t_1/x_1 \dots t_n/x_n] :: \tau)$$

Typregel zu Superkombinatoren und Anwendung:

Term $sc\ t_1 \dots t_n$:

Aus A herleitbar: $t_1 :: \tau_1, \dots, t_n :: \tau_n$,

ebenso $sc :: \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \in A$ und $(sc\ t_1 \dots t_n) :: \tau$.

Typannahmen konsistent \implies

$sc :: \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ mittels der Superkombinatorregel herleitbar
somit auch für den Rumpf $e :: \tau$ unter den Annahmen $x_1 :: \tau_1, \dots$

Typisierungsregeln beachten nur den Typ der Terme,
somit: $e[t_1/x_1 \dots t_n/x_n] :: \tau$ ist aus A herleitbar

2. $((\lambda x.s)t) \rightarrow s[t/x]$.

Analog zur Superkombinatorreduktion:

Abstraktions und Anwendungsregel-Typisierung

passen zur

Beta-Reduktion

3. case-Reduktion

$$\begin{aligned} & \text{case}_A (c_i t_1 \dots t_{n(i)}) && \{(c_1 x_{1,1}, \dots, x_{1,n(1)}) \rightarrow \text{exp}_1; \dots \\ & && (c_m x_{m,1}, \dots, x_{m,n(m)}) \rightarrow \text{exp}_m\} \\ & \longrightarrow && \\ & \text{exp}_i [t_1/x_{i,1}, \dots, t_{n(i)}/x_{i,n(i)}] \end{aligned}$$

case-Typregel:

\implies : $(c_i t_1 \dots t_{n(i)})$ und $(c_i x_{i,1} \dots x_{i,n(i)})$ haben gleichen Typ.

\implies : $t_i : \tau_i$ und $x_{j,i} : \tau_i \vdash \text{exp}_i : \tau$, wobei τ der Typ des case-Ausdrucks.

Beachte: Aus den Konventionen zu Konstruktoren folgt:
aus dem Typ von $(c_i t_1 \dots t_{n(i)})$

laesst sich der Typ der Terme t_i eindeutig bestimmen.

\implies : $\text{exp}_i [t_1/x_{i,1}, \dots, t_{n(i)}/x_{i,n(i)}] : \tau$

Definition: t dynamisch ungetypt:

gdw.

$t \xrightarrow{*} t'$ und t' direkt ungetypt

Lemma Für dynamisch ungetypte Ausdrücke lässt sich kein Typ herleiten.

Folgt aus obigen Lemmas:

1. Typ bleibt erhalten unter Reduktion
2. direkt dynamisch ungetypte t haben keinen Typ.

Satz

**wohlgetypte Programme machen
keine dynamischen Typfehler**

Das gilt auch für die (Milner-)Typisierung

Die Milner-Typisierung typisiert aber **weniger** Programme als die Grundtypenmethode.

Als Beispiel die Beta-Reduktion: $\frac{((\lambda x.t) s)}{t[s/x]}$.

Typen vorher: $\lambda x.t :: \tau_1 \rightarrow \tau_2 \quad s :: \sigma.$
 $((\lambda x.t) s) :: \gamma(\tau_2).$ wobei $\gamma(\tau_1) = \gamma(\sigma)$
Typ nachher: $t[s/x] :: ??$

Da der Typcheck-Algorithmus nur auf den Typ der Unterterme achtet und $s :: \gamma(\sigma)$ und $x :: \gamma(\sigma)$, gilt: $t[s/x] : \gamma(\tau_2)$

Analog für andere Auswertungsregeln

Progress-Lemma; Fortschrittslemma: (zB in KFPTSP)

Für jeden geschlossenen, wohlgetypten Ausdruck t gilt:
Entweder

- t ist eine WHNF; oder
- t hat eine Normalordnungsreduktion

Fortschrittslemma gilt **nicht** in KFPT für die erlaubten Ausdrücke

zB `(Cons Nil Nil) Nil`

ist keine WHNF, und hat keine Normalordnungsreduktion