Theorie der Programmierung

SoSe 2015

Dieses Skript zur Vorlesung *„Theorie der Programmierung“* (Prof. Dr. Lutz Schröder) wurde im Sommersemester 2014 von den untenstehenden Studenten erarbeitet und vom Veranstalter ab Sommersemester 2015 nur oberflächlich überarbeitet. Es ist inoffiziell und erhebt weder einen Anspruch auf Korrektheit noch auf Vollständigkeit.

Florian Jung `florian.jung@fau.de`

Christian Bay `christian.bay@studium.fau.de`

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

- Was tut ein Programm?
  - Terminiert es?
  - Liefert es korrekte Ergebnisse?

Plan:

- Termersetzung
- λ-Kalkül (LISP)
- Semantik von Programmiersprachen, Bereichstheorie, (Ko-)Datentypen, (Ko-)Induktion
- reguläre Ausdrücke

1.1 Literatur

Termersetzung:

- TES: Baader/Nipkow: Term Rewriting and all that
- Klop: Term rewriting systems
- Giesl: TES (Skript RWTH Aachen)

λ-Kalkül:

- Barendregt: λ-calcul: with types
- Skript TUM Nipkow

Semantik:

- Winskel: Formal Semantics of Programming Languages

reguläre Ausdrücke:

- Hopcroft/Ullmann/Motwani
- Pitts: Lecture Notes on Regular Languages and Finite Automata, Cambridge University, 2013

(Ko-)Induktion

- Jacobs/Rutten: A Tutorial on (Co-)Algebras and (Co-)Induction
- J. Rutten: Automata and Coinduction - an exercise in coalgebra

1.2 Konventionen

Natürliche Zahlen $0 \in \mathbb{N}$

Logische Implikation Für den Folge- und Äquivalenzpfeil werden die Symbole „$\Rightarrow$“ sowie „$\Leftrightarrow$“ genutzt. Das Symbol „$\rightarrow$“ ist für andere Verwendungen reserviert.

2 Termersetzung

Unter *Termersetzung* verstehen wir die sukzessive (und erschöpfende) Umformung von Termen gemäß *gerichteter* Gleichungen.

Anwendungen:

- (Algebraische) Spezifikation
- Programmverifikation
- automatisches Beweisen
- Programmierung
  - Turing-vollständig
  - Funktionale Programmiersprachen
- Computeralgebra (Gröbnerbasen / Buchbergeralgorithmus)

Beispiel 2.1 (Addition in Haskell)**.**

```
1 data Nat = Zero | Suc(Nat)
2 plus Zero y = y
3 plus (Suc x) y = Suc(plus x y )
```

(Der Datentyp *Nat* enthält damit Terme *Zero*, *Suc*(*Zero*), *Suc*(*Suc*(*Zero*)) etc.)

Beispiel 2.2 (Auswertung von $2 + 1$)**.**

$$\begin{aligned} plus\ (Suc\ (Suc\ (Zero)))(Suc\ (Zero)) &\rightarrow Suc\ (plus\ (Suc\ Zero)(Suc\ Zero)) \\ &\rightarrow Suc\ (Suc\ (plus\ Zero(Suc\ Zero))) \\ &\rightarrow Suc\ (Suc\ (Suc\ Zero)) \end{aligned}$$

Beispiel 2.3 („Assoziativgesetz“)**.** Behauptung: $(2 + x) + y = 2 + (x + y)$.

Beweis:

$$\begin{aligned} plus\ (Suc\ (Suc\ x))y &\rightarrow Suc\ (plus\ (Suc\ x)y) \\ &\rightarrow Suc\ (Suc\ (plus\ x\ y)) \end{aligned}$$

Beispiel 2.4 (Optimierung)**.**

$$plus\ (plus\ xy)z = plus\ x(plus\ yz)$$

Bei Auswertung der linken Seite der Gleichung gemäß der Definition von *plus* werden $x + (x + y) = 2x + y$ Schritte benötigt, bei der Auswertung der rechten Seite nur $y + x$ Schritte.

Beispiel 2.5 (Eine problematische „Optimierung“)**.** Wir stellen uns vorübergehend vor, wir wollten auch mit dem Kommutativgesetz

$$plus\ x\ y = plus\ y\ x$$

umformen. Bei der Auswertung z.B. des Terms

$$plus\ (Suc\ Zero)\ Zero$$

bekommen wir dann (nicht unbedingt unlösbare) Probleme mit der Terminierung.

Beispiel 2.6 (Spezifikation und Verifikation)**.** Stellen wir uns einen Moment lang vor, zur Spezifikation unseres Additionsprogramms gehöre die Gleichung

$$plus\ (Suc\ (Suc\ Zero))\ x = plus\ (Suc\ Zero)\ (Suc\ x)$$

(so etwas steht natürlich in keiner sinnvollen Spezifikation). Wir können durch stumpfes Umformen zeigen, dass das Programm diese Gleichung erfüllt, müssen dieses Mal aber *beide* Seiten der Gleichung umformen:

$$\begin{aligned} plus\ (Suc\ (Suc\ Zero))\ x &\to Suc\ (plus\ (Suc\ Zero)\ x) \\ &\to Suc\ (Suc\ (plus\ Zero\ x)) \\ &\to Suc\ (Suc\ x) \end{aligned}$$

und

$$\begin{aligned} plus\ (Suc\ Zero)\ (Suc\ x) &\to Suc\ (plus\ Zero\ (Suc\ x)) \\ &\to Suc\ (Suc\ x). \end{aligned}$$

2.1 Syntax und operationale Semantik

2.1.1 Recall: Binäre Relationen

Definition 2.7. Eine *binäre Relation* zwischen zwei Mengen X und Y ist ein $R \subseteq X \times Y$. Man schreibt xRy, wenn $(x, y) \in R$.

Beispiel 2.8. Die „Kleiner-Gleich“-Relation auf den natürlichen Zahlen ist also eine Teilmenge $\leq\ \subseteq \mathbb{N} \times \mathbb{N}$, nämlich tautologischerweise die Teilmenge $\leq\ = \{(n, m) \mid n \leq m\}$.

Definition 2.9. Eine Relation $R \subseteq X \times X$ heißt

- *reflexiv*, wenn xRx für alle $x \in X$;
- *symmetrisch*, wenn für alle $(x, y) \in R$ (also xRy) auch yRx gilt;
- *transitiv*, wenn für alle xRy und yRz auch xRz gilt;
- eine *Präordnung*, wenn R reflexiv und transitiv ist.
- eine *Äquivalenzrelation* oder einfach eine *Äquivalenz*, wenn R reflexiv, transitiv und symmetrisch ist.

Definition 2.10 (Standardkonstruktionen auf Relationen)**.**

- *Gleichheit* („=“): $id = \{(x, x) | x \in X\} = \Delta$. Diese Relation ist offenbar eine Äquivalenz, die kleinste Äquivalenz auf X.
- *Verkettung* oder *Komposition* zweier Relationen $R \subseteq Y \times Z$ und $S \subseteq X \times Y$:

  $$R \circ S := \{(x, z) \mid \exists y \text{ mit } (x \,\mathrm{S}\, y \wedge yRz)\}.$$

  (Achtung: wie auch bei der Komposition von Funktionen verwenden wir die *applikative* Schreibweise, d.h. $R \circ S$ heißt *erst* S, *dann* R.) Wir definieren induktiv die n-fache Verkettung einer Relation mit sich selbst:

  $$R^0 := id, \qquad R^n := R \circ R^{n-1}.$$

- Die *Umkehrrelation* oder *Inverse* einer Relation $R \subseteq X \times Y$ ist die Relation

  $$R^- = \{(x, y) \mid yRx\} \subseteq Y \times X$$

  (Beispiel: $\leq^-$ ist $\geq$).

Lemma 2.11. *Für eine Relation $R \subseteq X \times X$ gilt:*

- *R ist reflexiv $\Leftrightarrow id \subseteq R$*
- *R ist symmetrisch $\Leftrightarrow R^- \subseteq R$ ($\Leftrightarrow R^- = R$)*
- *R ist transitiv $\Leftrightarrow R \circ R \subseteq R$*

Definition 2.12. Sei $R \subseteq X \times X$ eine Relation. Der *reflexive/symmetrische/transitive Abschluss* von R ist die kleinste reflexive/symmetrische/transitive Relation, die R enthält.

Wir haben folgende explizite Darstellungen der verschiedenen Abschlüsse von R:

- Reflexiver Abschluss: $R \cup id$
- Symmetrischer Abschluss: $R \cup R^-$

- Transitiver Abschluss:

$$\begin{aligned} R^+ &:= R \ \cup \ (R \circ R) \ \cup \ (R \circ R \circ R) \ \cup \ \ldots \\ &= \bigcup_{n=1}^{\infty} R^n \\ &(= \{(x,y) \mid \exists n \geq 1 \text{ mit } (x,y) \in R^n\}), \end{aligned}$$

  also

$$x \, R^+ \, y \Leftrightarrow \exists x_0, \ldots, x_{n+1}. \, x = x_0 R x_1 R \ldots R x_n R x_{n+1} = y$$

  ($n+1$ R-Schritte für $n \geq 0$, also mindestens einer).

- Transitiv-reflexiver Abschluss:

$$R^* = \bigcup_{n=0}^{\infty} R^n = R^+ \cup id = (R \cup id)^+,$$

  also

$$x \, R^* \, y \Leftrightarrow \exists x_0, \ldots, x_n. \, x = x_0 R x_1 R \ldots R x_{n-1} R x_n = y.$$

  ($n \geq 0$ R-Schritte).

Lemma 2.13 (Erzeugte Äquivalenz)**.** *Seien $R, S \subseteq X \times X$.*

1. *Wenn S symmetrisch ist, dann sind auch S^+ und S^* symmetrisch.*
2. *$(R \cup R^-)^*$ ist symmetrisch.*
3. *$(R \cup R^-)^*$ ist die von R erzeugte Äquivalenz.*

Beweis. Zu 1.: Wenn x über den Pfad $x_1, \ldots, x_n$ mit y in Relation steht, dann kann man in $(R \cup R^-)^*$ diesen Pfad auch in die entgegengesetzte Richtung ablaufen.
2., 3. folgen dann sofort. □

Beispiel 2.14. Sei $R \subseteq \mathbb{N} \times \mathbb{N}$ ist im folgenden die Relation, die jede positive natürliche Zahl mit ihrem unmittelbaren Vorgänger in Beziehung setzt:

$$R := \{(n+1, n) \mid n \in \mathbb{N}\}.$$

Dann

$$\begin{aligned} nR^+m &\iff n > m \\ nR^*m &\iff n \geq m \\ n(R \cup R^-)m &\iff |n-m| = 1 \\ n(R \cup R^-)^+m &\quad \text{stets (warum?)} \\ n(R \cup R^-)^*m &\quad \text{stets.} \end{aligned}$$

2.1.2 Recall: Terme

Definition 2.15.

- Eine *Signatur* Σ ist eine Menge von *Funktionssymbolen* $f, g, \ldots$ jeweils gegebener Stelligkeit. Wir schreiben $f/n \in \Sigma$, wenn f n-stelliges Funktionssymbol in Σ ist. Ein *Konstante* ist ein nullstelliges Funktionssymbol.
- Sei V eine Menge von *Variablen*. Ein *Term über* V ist dann induktiv definiert wie folgt:
  - Sei $x \in V$. Dann ist x, also eine einzelne Variable, ein Term.
  - Seien $t_1, \ldots, t_n$ Terme und sei $f/n \in \Sigma$ (d.h. f ist n-stellige Funktion). Dann ist $f(t_1, \ldots, t_n)$ ein Term. (Damit ist insbesondere jede Konstante ein Term.)

  M.a.W. sind Terme t über V definiert durch die Grammatik

$$t ::= x \mid f(t_1, \ldots, t_n) \qquad (x \in V, f/n \in \Sigma).$$

- Wir schreiben $T_\Sigma(V)$ für die Menge aller Terme.

Definition 2.16 (Freie Variablen)**.** Die Menge $FV(t)$ der *freien Variablen* in t ist rekursiv definiert durch

$$FV(x) = \{x\}$$
$$FV(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} FV(t_i).$$

(Es macht im Moment nur mäßig viel Sinn, von „freien“ Variablen zu reden, da es bisher keine gebundenen Variablen gibt; das kommt aber noch.)

Definition 2.17 (Substitution)**.** Eine *Substitution* ist eine Abbildung $\sigma : V_0 \to T_\Sigma(V)$ für eine endliche Teilmenge $V_0 \subseteq V$, d.h. eine Vorschrift zur Ersetzung von Variablen durch Terme. Wenn nichts anderes gesagt ist, nehmen wir typischerweise an, dass V_0 nur „relevante“, d.h. in den Termen, auf die wir σ anwenden, tatsächlich vorkommende Variablen enthält. Wir schreiben

$$[t_1/x_1, \ldots, t_n/x_n]$$

für die Substitution mit Definitionsbereich $\{x_1, \ldots, x_n\}$, die für $i = 1, \ldots, n$ jeweils x_i auf t_i abbildet.

Die *Anwendung* einer Substitution auf einen Term t wird $t\sigma$ geschrieben und ist rekursiv definiert durch

$$x\sigma = \sigma(x)$$
$$f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$$

Definition 2.18 (Kontext)**.**

1. Ein *Kontext* ist ein Term $C(\cdot)$ mit genau einem „Loch“ $(\cdot)$. Formal sind Kontexte definiert durch die Grammatik

$$C(\cdot) ::= (\cdot) \mid f(t_1, \ldots, t_{i-1}, C(\cdot), t_{i+1}, \ldots, t_n)$$

2. Das Resultat $C(t)$ der *Einsetzung* eines Terms t in einen Kontext $C(\cdot)$ ist rekursiv definiert durch

$$\begin{aligned}(\cdot)(t) &= t\\ f(t_1, \ldots, C(\cdot), \ldots, t_n)(t) &= f(t_1, \ldots, C(t), \ldots, t_n)\end{aligned}$$

Definition 2.19. Seien s, t Terme und σ eine Substitution.

1. Ein *Termersetzungssystem* ist eine Relation

$$\to_0 \subseteq T_\Sigma(V) \times T_\Sigma(V).$$

2. Eine Relation $R \subseteq T_\Sigma(V) \times T_\Sigma(V)$ heißt

   - *kontextabgeschlossen*, wenn

     $$tRs \implies C(t)RC(s)$$

     für jeden Kontext $C(\cdot)$ und alle Terme s, t;
   - *stabil*, wenn

     $$tRs \implies (t\sigma)R(s\sigma)$$

     für jede Substitution σ und alle Terme t, s.

3. Die *Einschrittreduktion* $\to \subseteq T_\Sigma(V) \times T_\Sigma(V)$ ist definiert als der kontextabgeschlossene stabile Abschluss von $\to_0$:

$$\to = \{(C(t\sigma), C(s\sigma)) \mid t \to_0 s, C(\cdot) \text{ Kontext}, \sigma \text{ Substitution}\}.$$

4. Die *Reduktionsrelation* oder einfach *Reduktion* ist der transitive-reflexive Abschluss $\to^*$ von $\to$. Ein Term t *reduziert* auf einen Term s, wenn $t \to^* s$.
5. *Konvertierbarkeit* $\leftrightarrow^* = (\to \cup \to^-)^*$ ist die von $\to$ erzeugte Äquivalenz.
6. Ein Term t heißt *normal*, wenn t nicht reduziert werden kann, d.h. wenn kein Term s mit $t \to s$ existiert. Schreibweise: $t \not\to$.
7. Ein Term s heißt eine *Normalform* eines Terms t, wenn s normal ist und $t \to^* s$.

Beispiel 2.20. Wir definieren $\to_0$ durch

$$x + (y + z) \to_0 (x + y) + z$$

(d.h. wir verwenden die Signatur $\Sigma = \{+/2\}$ und setzen $\to_0 = \{(x + (y + z), (x + y) + z)\}$; wir bleiben ab jetzt bei der obigen lesbareren Schreibweise).

Dann haben wir

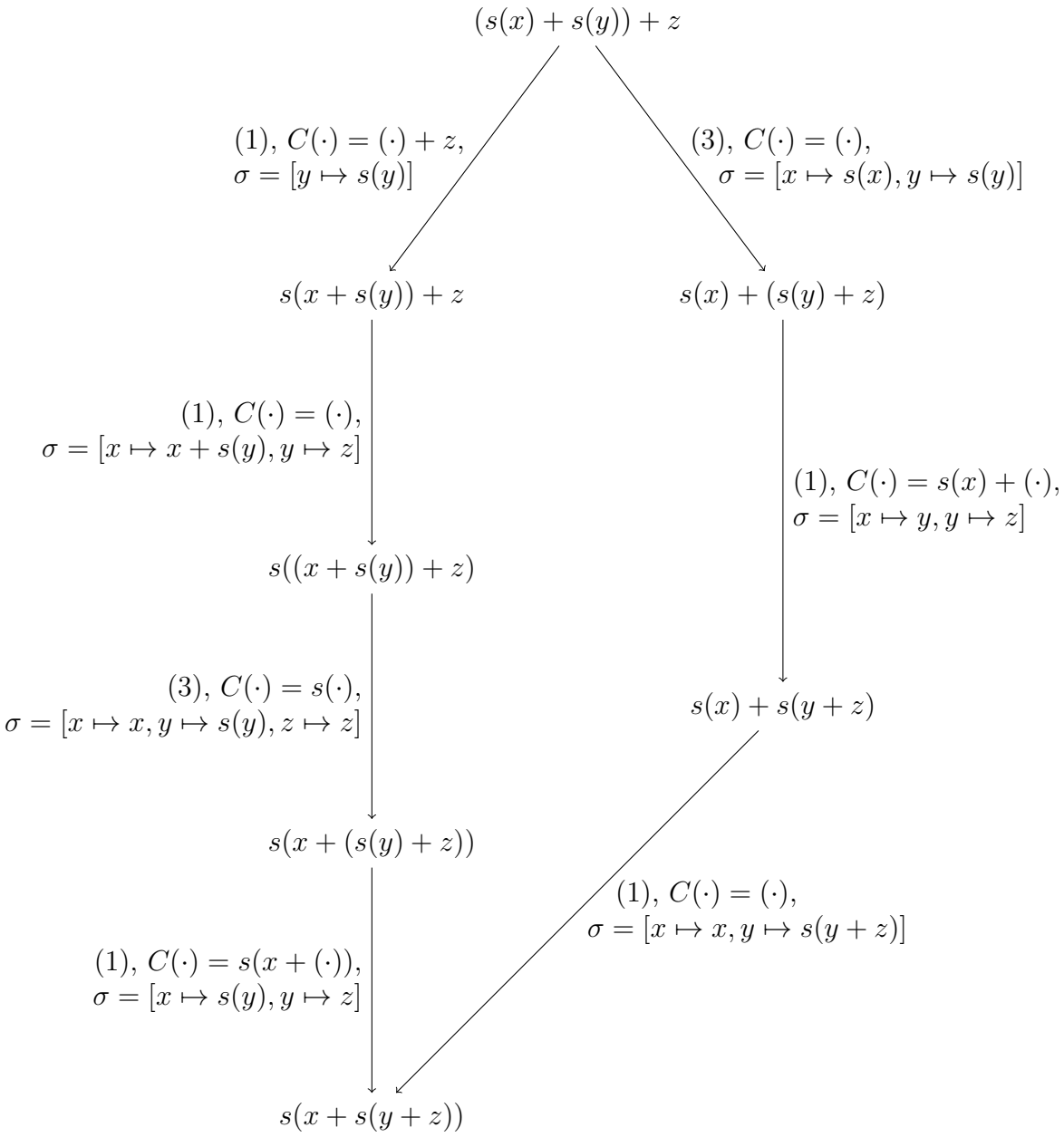
$$(a + (b + (c + d))) + e \to ((a + b) + (c + d)) + e.$$

Hierbei verwenden wir als Kontext $C(\cdot) = (\cdot) + e$ und als Substitution $\sigma = [x \mapsto a, y \mapsto b, z \mapsto c + d]$.

Beispiel 2.21. Sei $\Sigma = \{+/2, s/1, 0/0\}$ und $\to_0$ definiert durch

$$s(x) + y \to_0 s(x + y) \tag{1}$$
$$0 + y \to_0 y \tag{2}$$
$$(x + y) + z \to_0 x + (y + z) \tag{3}$$

Wir formen nun den Ausdruck $(s(x) + s(y)) + z$ um. Wir stellen fest, dass wir zwei verschiedene Regeln anwenden können:



Der letzte Term ist eine Normalform. Wir sehen, dass es *in diesem Fall* keine Rolle spielt, welche Wahl wir am Anfang treffen; beide Reduktionen führen hier auf dieselbe Normalform.

2.2 Terminierung

Definition 2.22. Eine Relation $R \subseteq X \times X$ heißt *wohlfundiert* (well-founded, wf.) genau dann wenn es *keine* unendliche Folge $(x_i)_{i \in \mathbb{N}}$ gibt mit $x_0 R x_1 R x_2 \ldots$.

Beispiel 2.23. $(\mathbb{N}, >)$ ist wohlfundiert, $(\mathbb{Z}, >)$ nicht.

Sei $(\Sigma, \to_0)$ ein Termersetzungssystem.

Definition 2.24. Ein Term t heißt

- *schwach normalisierend*, wenn t eine Normalform hat, d.h. wenn s existiert mit $t \to^* s$ und s normal;
- *stark normalisierend*, wenn *keine* unendliche Folge $(t_i)_{i\in\mathbb{N}}$ mit $t = t_0 \to t_1 \to t_2 \to \ldots$ existiert

Ein Termersetzungssystem $(\Sigma, \to_0)$ heißt *schwach/stark normalisierend (WN/SN)*, wenn jeder Term schwach/stark normalisierend in $\to$ ist. (Insbesondere ist also $(\Sigma, \to_0)$ SN, wenn $\to$ wohlfundiert ist.)

Beispiel 2.25. Wir definieren $\to_0$ durch

$$f(x) \to_0 f(x)$$
$$g(x) \to_0 1.$$

- $g(x)$ ist stark normalisierend: $g(x) \to 1 \not\to$.
- $f(3)$ ist nicht schwach normalisierend: $f(3) \to f(3) \to f(3) \to \ldots$.
- $g(f(3))$ ist schwach normalisierend: $g(f(3)) \to 1 \not\to$
- $g(f(3))$ ist nicht stark normalisierend: $g(f(3)) \to g(f(3)) \to \ldots$

2.2.1 Terminierung

Definition 2.26.

1. Eine Relation $R \subseteq X \times X$ heißt *irreflexiv* genau dann, wenn für alle x gilt: $\neg(xRx)$.
2. R heißt *strikte Ordnung*, wenn R irreflexiv und transitiv ist.
3. $R \subseteq T_\Sigma(v) \times T_\Sigma(V)$ heißt *Reduktionsordnung*, wenn R eine stabile, kontextabgeschlossene, wohlfundierte, strikte Ordnung ist.

Zur Erinnerung: R ist kontextabgeschlossen, wenn mit tRs stets auch $C(t)RC(t)$ gilt; wir merken an, dass man hierbei offenbar auf Kontexte $C(\cdot)$ von der Form $C(\cdot) = f(x_1, \ldots, (\cdot), \ldots, x_n)$ einschränken darf.

Satz 2.27. *Sei $>$ Reduktionsordnung, und für alle Terme t, s gelte: Aus $t \to_0 s$ folgt $t > s$. Dann ist $\to_0$ stark normalisierend.*

Beweis. Es gilt auch $\forall t, s : t \to s \Rightarrow t > s$, da $>$ kontextabgeschlossen und stabil ist; d.h. $\to$ ist Teilrelation der wohlfundierten Relation $>$ und somit selbst wohlfundiert. □

Zwei offensichtliche Ideen zur Definition von Reduktionsordnungen klappen im allgemeinen leider nicht:

Beispiel 2.28. Sei $|t|$ die Größe von t. Betrachte die durch $t > s :\Leftrightarrow |t| > |s|$ definierte Relation $>$.

- $>$ ist kontextabgeschlossen: $|C(t)|$ ist im wesentlichen $|C(\cdot)| + |t|$, also folgt $|C(t)| > |C(s)|$ aus $|t| > |s|$.
- $>$ ist i.a. nicht stabil: $(x + 2y) - x > y + y$, aber $(x + 2t) - x \not> t + t$ für t groß.

(Stabilität haben wir aber offenbar dann, wenn in jeder Reduktion $l \to_0 r$ alle Variablen höchstens so oft in r vorkommen wie in l.)

Beispiel 2.29. Definiere $>$ durch $t > s :\Leftrightarrow s$ ist echter Unterterm von t. Damit ist $>$ wohlfundiert und stabil, aber nicht kontextabgeschlossen: Es gilt $x + x > x$, aber nicht $f(x + x) > f(x)$.

Die folgende Definitionen dagegen klappen immer, sind aber eher nutzlos:

Beispiel 2.30. $\emptyset$ ist Reduktionsordnung.

Beispiel 2.31. $\to^+$ ist Reduktionsordnung, wenn $\to_0$ stark normalisierend.

2.3 Polynomordnungen

2.3.1 Recall: Polynome

Polynome über $\mathbb{N}$:

$$\mathbb{N}[X_1, \ldots, X_N] = \left\{ \sum a_{i_1,\ldots,i_n} \cdot X_1^{i_1} \cdot \cdots \cdot X_n^{i_n} \mid a_{i_1,\ldots,i_n} = 0 \text{ fast immer} \right\};$$

z.B. $X^2Y + 2Y^2ZX \in \mathbb{N}[X, Y, Z]$.

Summen und Produkte von Polynomen sind Polynome (letztere nach Ausmultiplizieren), also

$$p(q_1, \ldots, q_n) \in \mathbb{N}[Y_1, \ldots, Y_m]$$

für $p \in \mathbb{N}[X_1, \ldots, X_n]$ und $q_1, \ldots, q_n \in \mathbb{N}[Y_1, \ldots, Y_m]$ (wobei wir mit $p(q_1, \ldots, q_n)$ das Polynom bezeichnen, das durch Einsetzen der Polynome q_i für die Variablen in p entsteht).

Definition 2.32. Für $\emptyset \neq A \subseteq \mathbb{N}$ definieren wir eine Ordnung $>_A$ auf $\mathbb{N}[X_1, \ldots, X_n]$ durch

$$p >_A q \Leftrightarrow \forall k_1, \ldots, k_n \in A.\, p(k_1, \ldots, k_n) > q(k_1, \ldots, k_n).$$

Beispiel: $X^2 >_{\mathbb{N}_{\geq 2}} X$.

Lemma 2.33. *$>_A$ ist wohlfundiert.*

Beweis. Wähle $a \in A$. Angenommen, es gäbe eine Folge $(p_i)_{i\in\mathbb{N}}$ mit $p_0 >_A p_1 >_A \ldots$. Dann gälte $\underbrace{p_0(a, \ldots, a)}_{\in\mathbb{N}} > p_1(a, \ldots, a) > \ldots$, was in $\mathbb{N}$ aber unmöglich ist. □

Definition 2.34. Ein Polynom $p \in \mathbb{N}[X_1, \ldots, X_n]$ heißt *streng monoton*, wenn jedes X_i in p vorkommt, formal: $\forall j \in \{1, \ldots, n\}. \exists i_1, \ldots, i_n \geq 0. i_j > 0 \wedge a_{i_1,\ldots,i_n} > 0$.

Lemma 2.35. *Wenn p streng monoton im obigen formalen Sinn ist, dann ist p streng monoton im üblichen Sinn, d.h. wenn $k_1, \ldots, k_n, l_1, \ldots, l_n \in \mathbb{N}$ mit $k_j \geq l_j$ für alle j und $k_j > l_j$ für mindestens ein j, dann gilt $p(k_1, \ldots, k_n) > p(l_1, \ldots, l_n)$.*

Definition 2.36. Eine *(monotone) polynomielle Interpretation* $\mathcal{A}$ für Σ besteht aus

- einem streng monotonen Polynom $p_f \in \mathbb{N}[X_1, \ldots, X_n]$ für jedes $f/n \in \Sigma$
- einer Teilmenge $A \subseteq \mathbb{N}$, die unter den p_f *abgeschlossen* ist, d.h. $p_f(a_1, \ldots, a_n) \in A$ für alle $a_1, \ldots, a_n \in A$ und alle $f/n \in \Sigma$.

Die hierdurch induzierte *Polynomordnung* auf Termen ist definiert durch

$$t \succ_{\mathcal{A}} s :\Leftrightarrow p_t >_A p_s$$

für Terme t, s, wobei p_t rekursiv definiert ist durch

$$p_x = X.$$
$$p_{f(t_1,\ldots,t_n)} = p_f(p_{t_1}, \ldots, p_{t_n}).$$

Satz 2.37. *Polynomordnungen sind Reduktionsordnungen.*

Beweis.

- $\succ$ wohlfundiert, da $>_A$ wohlfundiert.
- $\succ$ stabil: klar
- $\succ$ kontextabgeschlossen: per p_f streng monoton.

□

Korollar 2.38. *Wenn $\mathcal{A}$ eine Polynomordnung ist, so dass*

$$t \to_0 s \implies t \succ_{\mathcal{A}} s \quad \textit{für alle } t, s,$$

dann ist T SN.

Beweis. Unmittelbar mit Satz 2.37 und Satz 2.27. □

Beispiel 2.39. Wir definieren ein TES $\to_0$ durch

$$(x \oplus y) \oplus z \to_0 x \oplus (y \oplus z) \tag{4}$$
$$x \oplus (y \oplus z) \to_0 y \oplus y. \tag{5}$$

Wir verwenden die durch $A = \mathbb{N}_{\geq 3}$ und

$$p_{\oplus}(X, Y) = X^2 + XY$$

gegebene Polynomordnung. Für Reduktion (4) rechnen wir

$$\begin{aligned} p_{(x\oplus y)\oplus z} &= p_\oplus(p_\oplus(X,Y),Z) \\ &= p_\oplus(X^2+XY,Z) \\ &= (X^2+XY)^2+(X^2+XY)Z \\ &= X^4+4X^3Y+X^2Y^2+X^2Z+XYZ \end{aligned}$$

sowie

$$\begin{aligned} p_{x\oplus(y\oplus z)} &= p_\oplus(X,p_\oplus(Y,Z)) \\ &= X^2+X(Y^2+Z) \\ &= X^2+XY^2+XZ, \end{aligned}$$

so dass offenbar $p_{(x\oplus y)\oplus z} >_A p_{x\oplus(y\oplus z)}$. Ferner haben wir

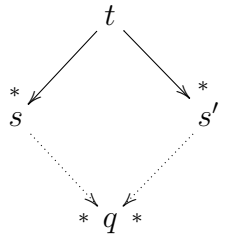
$$p_{y\oplus y} = Y^2+Y^2 = 2Y^2,$$

so dass offenbar $p_{x\oplus(y\oplus z)} >_A p_{y\oplus y}$. Nach Korollar 2.38 ist $\to_0$ somit SN.

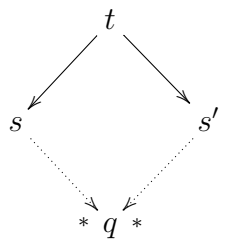
2.4 Konfluenz

Definition 2.40. Sei T ein Termersetzungssystem.

1. Terme s, s' heißen *zusammenführbar* (zf.), wenn ein q existiert mit $s \to^* q$ und $s' \overset{*}{\to} q$.
2. Das Termersetzungssystem T heißt *konfluent* (CR, nach Church-Rosser), wenn für alle Terme t, s, s' mit $t \overset{*}{\to} s$ und $t \overset{*}{\to} s'$ die Terme s und s' sind zusammenführbar sind:



3. Das Termersetzungssystem T heißt *lokal konfluent* (WCR), wenn für alle Terme t, s, s' mit $t \to s$ und $t \to s'$ die Terme s und s' zusammenführbar sind:



Fakt 2.41. *Syntaktisch gleiche Terme sind zusammenführbar.*

Bemerkung 2.42. Konfluenz ist eine wichtige Eigenschaft insbesondere dann, wenn man TES als Programme ansieht, da sie Determinismus garantiert: Wenn ein Termersetzungssystem konfluent ist, mag es zwar eventuell mehrere mögliche Umformungswege geben, doch sie alle werden (falls sie terminieren) zu dem gleichen Ergebnis führen. Wenn man (in der Computer-Algebra und anderswo) Termersetzung zur Analyse von Gleichungstheorien verwendet, folgt, wie wir sehen werden, aus CR und SN Entscheidbarkeit der durch das TES dargestellten Gleichungstheorie.

Satz 2.43. *Sei T ein konfluentes Termersetzungssystem.*

1. *Für Terme t, s gilt*

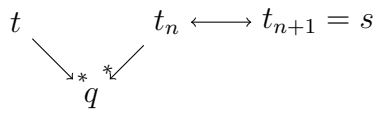
$$s \leftrightarrow^* t \iff s \text{ und } t \text{ sind zusammenführbar.}$$

2. *Normalformen sind eindeutig, d.h. wenn Terme s, s' Normalformen eines Terms t sind, dann gilt $s = s'$ (d.h. s und s' sind syntaktisch gleich).*

Beweis.

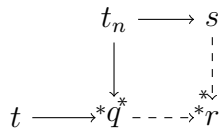
(1.) „$\Leftarrow$“ ist klar; wir zeigen „$\Rightarrow$“: Nach Voraussetzung existieren $n \geq 0$ und $t_1, \ldots, t_n$ mit $t = t_0 \leftrightarrow t_1 \leftrightarrow \ldots \leftrightarrow t_n = s$. Induktion über n:

- $n = 0$: Dann gilt $s = t$, also sind t und s nach Fakt 2.41 zusammenführbar.
- $n \to n + 1$: Nach Induktionsvoraussetzung haben wir:



  Fall 1: $s \to t_n$. Dann $s \xrightarrow{*} q$, d.h. t und s sind zusammenführbar.

  Fall 2: $t_n \to s$. Dann existiert per Konfluenz r mit $q \xrightarrow{*} t$ und $s \xrightarrow{*} r$. Es gilt dann auch $t \xrightarrow{*} r$, d.h. t und s sind zusammenführbar:



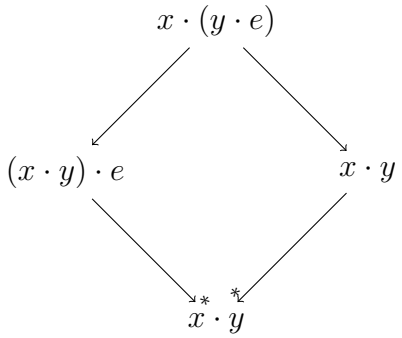
(2.) Nach Voraussetzung $s \leftrightarrow^* s'$, also gibt es nach (1.) ein q mit $s \xrightarrow{*} q \leftarrow^* s'$, also $s = q = s'$, da s, s' normal sind.

□

Beispiel 2.44 (Gruppen)**.** Wir formulieren die Definition einer Gruppe mittels der Verknüpfung $\cdot$, des neutralen Elements e und der Inversenbildung $\_^{-1}$. In dieser Signatur drücken wir die Gruppenaxiome durch das TES

$$\begin{aligned} x \cdot (y \cdot z) &\to_0 (x \cdot y) \cdot z \\ x \cdot e &\to_0 x \\ x \cdot x^{-1} &\to_0 e \end{aligned}$$

aus. Damit kann z.B. der Ausdruck $x \cdot (y \cdot e)$ zunächst zu zwei unterschiedlichen Ausdrücken umgeformt werden. Allerdings führen beide Umformungen letztlich auf dieselbe Normalform:



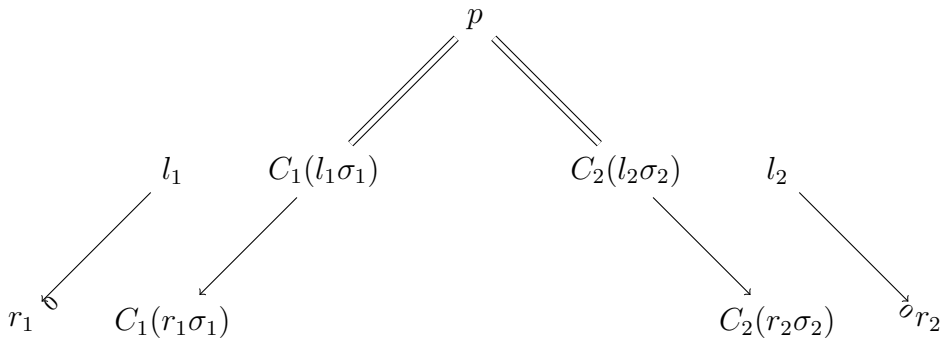
Die Bedeutung der lokalen Konfluenz liegt in der folgenden Tatsache:

Satz 2.45 (Newman's Lemma)**.** *Ein* stark normalisierendes *und* lokal konfluentes *Termersetzungssystem ist* konfluent*.*

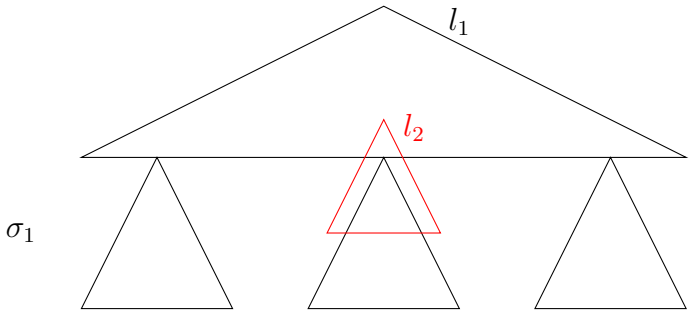
Der Beweis folgt auf Seite 26.

Wir wenden uns nun der Frage zu, wann ein Termersetzungssystem *lokal konfluent* (WCR) ist. Eine zentrale Rolle spielt hierbei der Begriff des *kritischen Paar* nach Knuth und Bendix.

Betrachten wir die folgende Situation, in dem ein Term p mit zwei verschiedenen Regeln reduziert werden kann:



Derartig konkurrierende Reduktionsmöglichkeiten werden insbesondere dann zum Problem, wenn die Anwendung der zweiten Regel $l_2 \to_0 r_2$ im Termbaum unterhalb der von $l_1 \to_0 r_1$ liegt, also unter dem Kontext $C_1(\cdot)$, und somit durch ihre Anwendung die Anwendbarkeit der ersten Regel gestört wird. Die folgende Grafik zeigt einen Ausdruck l_1 zusammen mit einer Substitution σ_1, der als solcher zu $r_1\sigma_1$ reduziert werden könnte. Allerdings ragt l_2 (rot) in l_1 hinein. Nach Anwendung von $l_2 \to_0 r_2$ liegt dann kein mit l_1 unifizierbarer Ausdruck mehr vor, $l_1 \to_0 r_1$ ist also nicht mehr anwendbar.



(6)

Für eine formale Definition solcher Situationen erinnern wir an einen Begriff aus GLoIn:

Definition 2.46. Eine Substition σ heißt *Unifikator* von Termen t, s, wenn $t\sigma = s\sigma$. Wir setzen $\mathit{Unif}(t, s) := \{\sigma \mid \sigma \text{ ist Unifikator von } t, s\}$. Terme t, s heißen *unifizierbar*, wenn $\mathit{Unif}(t, s) \neq \emptyset$.

Beispiel 2.47.

1. Die Terme $g(x, h(x))$ und $g(h(y), y)$ sind nicht unifizierbar: Ein Unifikator σ müsste sowohl $h(x) = y$ als auch $x = h(y)$ lösen.
2. Die Terme $f(g(x, y), z)$ und $f(v, h(w))$ sind unifizierbar mit $\sigma := [v \mapsto g(x, y), z \mapsto h(w)]$

Definition 2.48. Eine Substitution σ heißt *allgemeiner als* eine Substitution σ', wenn eine Substitution τ mit $\sigma' = \sigma\tau$ existiert. Wir sagen, σ sei *allgemeinster Unifikator* von Termen t, s, und schreiben $\sigma = \mathit{mgu}(t, s)$, wenn σ Unifikator von t, s ist sowie allgemeiner als alle Unifikatoren von t, s.

Damit formulieren wir den Begriff des kritischen Paars wie folgt.

Definition 2.49. Seien $l_1 \to_0 r_1$ und $l_2 \to_0 r_2$ zwei Umformungsregeln des Termersetzungssystems sowie $FV(l_1) \cap FV(l_2) = \emptyset$ (ggf. nach Umbenennung). Sei $l_1 = C(t)$, wobei t ein nichttrivialer Term ist (d.h. t ist nicht nur eine Variable), so dass t und l_2 unifizierbar sind. Sei $\sigma = \mathit{mgu}(t, l_2)$. Dann heißt $(r_1\sigma, C(r_2)\sigma)$ ein *kritisches Paar*.

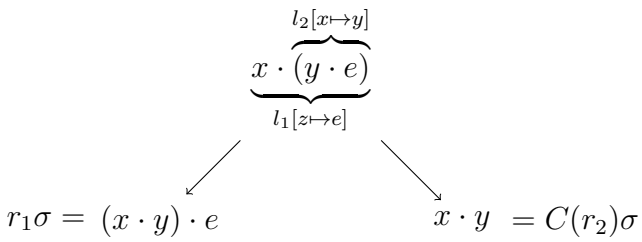
Beispiel 2.50 (Gruppen, siehe oben)**.** Wir vergeben Bezeichner für die in $\to_0$ vorkommenden Terme:

$$(l_1 \to_0 r_1) = (x \cdot (y \cdot z) \to_0 (x \cdot y) \cdot z)$$
$$(l_2 \to_0 r_2) = (x' \cdot e \to_0 x').$$

Dann haben wir ein kritisches Paar $(r_1\sigma, C(r_2\sigma))$, wobei $\sigma = mgu(t, l_2)$ mit

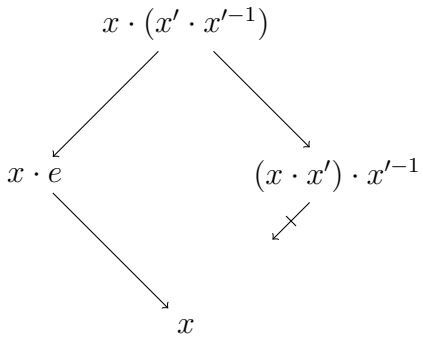
$$t = y \cdot z \qquad C(\cdot) = x \cdot (\cdot),$$

also $\sigma = [y \mapsto x', z \mapsto e]$:



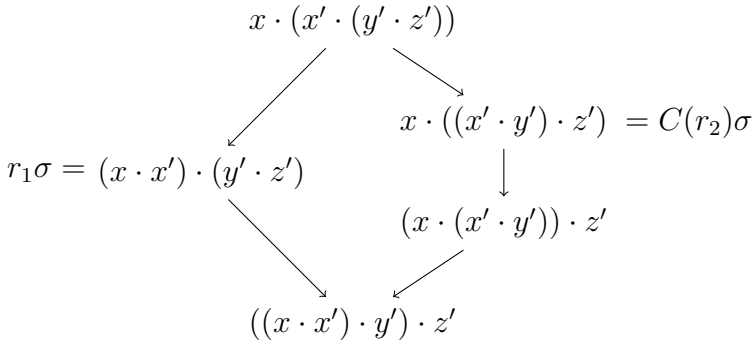
(Wir haben oben gesehen, dass dieses kritische Paar zusammenführbar ist.)

Beispiel 2.51. Mit $\sigma = mgu(y \cdot z, x' \cdot x'^{-1}) = [x'/y, x'^{-1}/z]$ haben wir (immer noch im obigen TES für Gruppen):



d.h. wir sind auf ein nicht zusammenführbares kritisches Paar $(x \cdot e, (x \cdot x') \cdot x'^{-1})$ gestoßen. Um ein konfluentes System zu erhalten, müssen wir also noch eine weitere Regel hinzufügen.

Beispiel 2.52. Achtung: die linke Seite einer Regel kann durchaus auch mit einem ihrer Teilterme unifizieren, was dann ein kritisches Paar ergibt. Z.B. können wir die linke Seite $x \cdot (y \cdot z)$ des Assoziativgesetzes mit $y' \cdot z'$ unifizieren: wir haben $\sigma = mgu(y \cdot z, x' \cdot (y' \cdot z')) = [x'/y, y' \cdot z'/z]$. Das entstehende kritische Paar ist zusammenführbar:



Bemerkung 2.53. Es gibt (für $\to_0$ endlich) nur endlich viele kritische Paare.

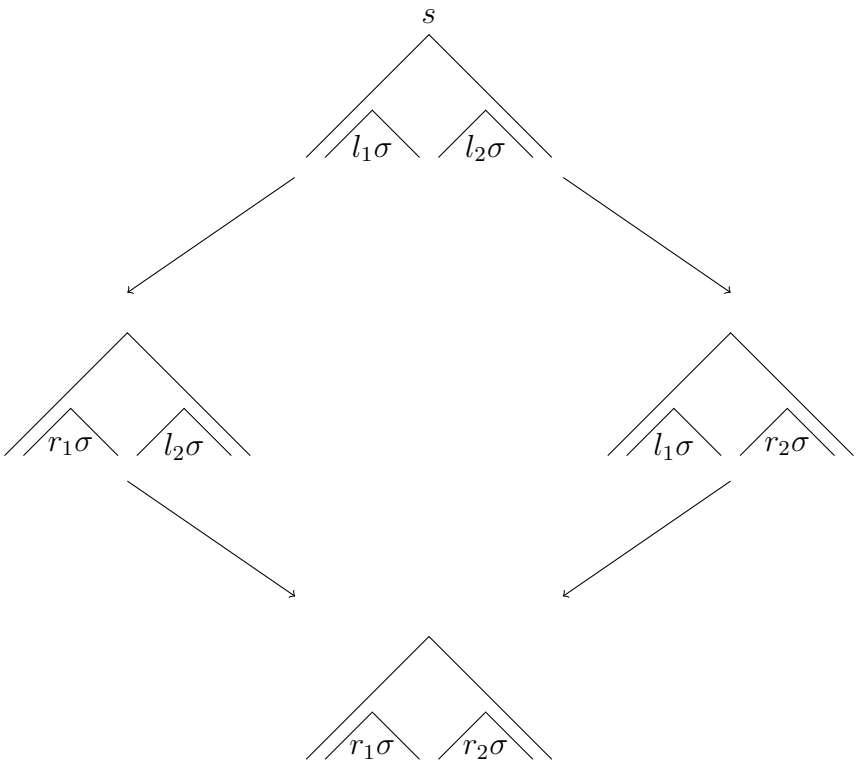
Satz 2.54 (Critical Pair Lemma)**.** *Ein Termersetzungssystem T ist genau dann lokal konfluent, wenn in T alle kritischen Paare zusammenführbar sind.*

Beweis. „Nur dann, wenn“ ist klar; wir beweisen die umgekehrte Implikation. Wir schreiben

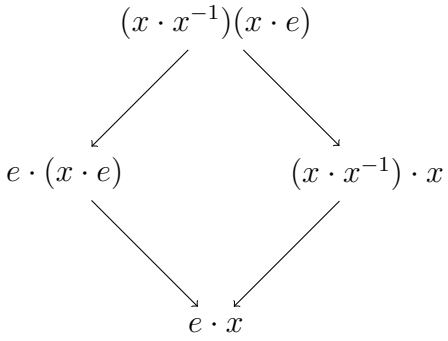
- $C(\cdot) \sqsubseteq D(\cdot)$, wenn E existiert mit $C(\cdot) = D(E(\cdot))$ (d.h. wenn das $(\cdot)$ von D im Syntaxbaum unterhalb von dem von C liegt).
- $C(\cdot) \perp D(\cdot)$, wenn $C(\cdot) \not\sqsubseteq D(\cdot)$ und $D(\cdot) \not\sqsubseteq C(\cdot)$.

Der Beweis läuft nun per Fallunterscheidung. Sei s ein Term, auf den Regeln $l_1 \to r_1$ und $l_2 \to r_2$ anwendbar sind, mit Kontext $C_1(\cdot)$ bzw. $C_2(\cdot)$ und Substitution σ_1 bzw. σ_2.

Fall 1: $C_1(\cdot) \perp C_2(\cdot)$: In diesem Fall liegen die Regelanwendungen in disjunkten Teiltermen von s, und sind daher zusammenführbar, da sie sich gegenseitig nicht stören:

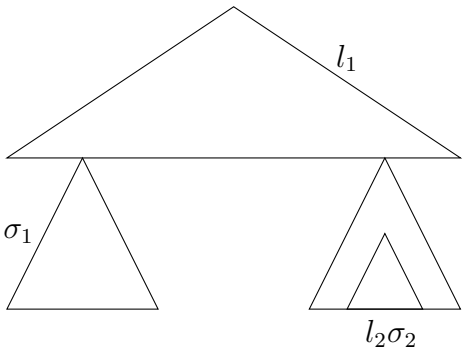


Beispiel: In unserem TES für Gruppen haben wir



Fall 2: O.E. $C_2(\cdot) \sqsubseteq C_1(\cdot)$ und $C_1(\cdot) = (\cdot)$; d.h. $C_2(l_2\sigma_2) = l_1\sigma_1$. Wir unterscheiden wiederum zwei Fälle:

Fall 2a: die Anwendung der zweiten Regel liegt unterhalb von l_1, d.h. $C_2(\cdot)$ entsteht aus einem Term der Form $l_1\sigma'$ durch Ersetzen eines einzelnen Vorkommens einer Variable durch $(\cdot)$:



O.E. nehmen wir $\sigma_2 = id$ an (ein nichttriviales σ_2 kann man sonst einfach hinterher wieder in die gesamte Rechnung einsetzen). Wir müssen zeigen, dass $C_2(r_2)$ und $r_1\sigma_1$ zusammenführbar sind. Wir haben gesehen, dass r_2 innerhalb eines Teilterms $\sigma_1(x)$ liegt; wenn x in l_1 mehrfach vorkommt, ist die erste Regel jetzt zunächst nicht mehr anwendbar. Wir stellen Anwendbarkeit der ersten Regel wieder her, indem wir die zweite Regel auch für alle anderen Vorkommen von x auf den entsprechenden Teilterm $\sigma_1(x)$ in $C_2(r_2)$ anwenden. Wir erreichen dann wieder einen Term der Form $l_1\sigma'$, auf den wir die erste Regel anwenden können. Auf $r_1\sigma_1$ können wir sofort die zweite Regel anwenden (wiederum so oft, wie x in r_1 vorkommt), da ja l_2 innerhalb von σ_1 liegt; damit erreichen wir denselben Term wie vorher, haben also Konfluenz gezeigt. Abbildung 1 illustriert den Fall, dass x in l_1 dreimal vorkommt.

Fall 2b: In diesem Fall ragt l_2 in l_1 hinein wie in (6) oben illustriert; wir haben also ein kritisches Paar, das nach Voraussetzung zusammenführbar ist.

□

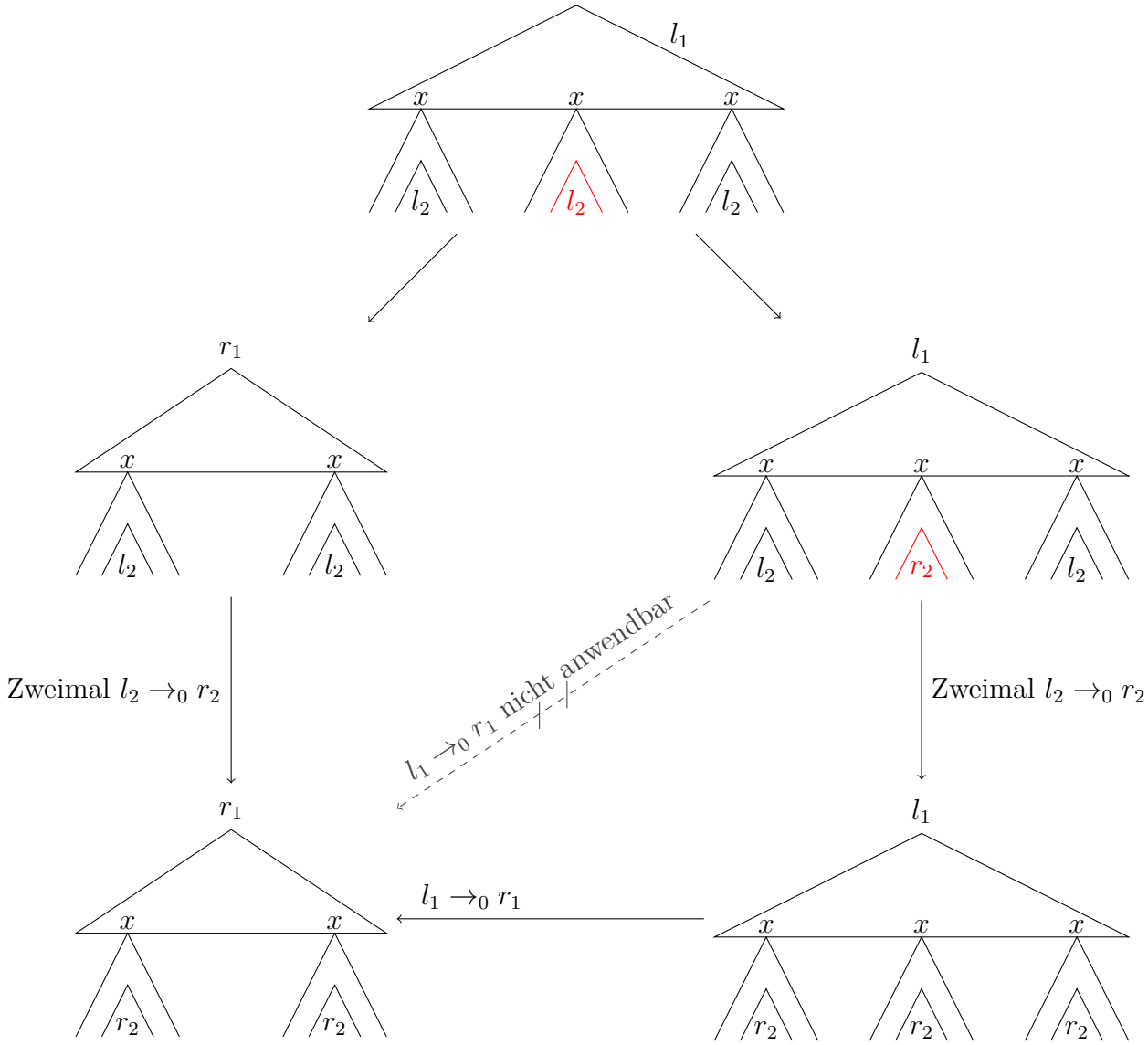


Abbildung 1: Konfluenz in Fall 2a

2.5 Wohlfundierte Induktion

Zur Vervollständigung des Bilds fehlt uns noch der Beweis von Newman's Lemma. Dieser verwendet Induktion über wohlfundierte Relationen:

Satz 2.55. *Sei $R \subseteq X \times X$ eine wohlfundierte Relation auf einer Menge X. Dann gilt folgendes Prinzip der* wohlfundierten Induktion*:*
(*) Falls gilt: $\forall x.\, (\forall y.\, x \,\mathrm{R}\, y \Rightarrow P(y)) \Rightarrow P(x)$
(**) Dann folgt: $\forall x.\, P(x)$

Dabei nennen wir (**) die *Induktionsbehauptung*, die Implikation (*) den *Induktionsschritt*, und die im Induktionsschritt verwendete Annahme $P(y)$ für alle y mit xRy die *Induktionsvoraussetzung*.

Beweis. Angenommen, (**) gelte nicht, d.h. es existiert x_0 mit $\neg P(x_0)$. Dann gibt es wegen (*) ein x_1 mit $x_0 \,\mathrm{R}\, x_1$, das die Induktionsvoraussetzung nicht erfüllt, d.h. $\neg P(x_1)$.
Iterieren gibt $x_0 \,\mathrm{R}\, x_1 \,\mathrm{R}\, x_2 \,\mathrm{R} \ldots$, im Widerspruch zur Wohlfundiertheit von R. □

Beispiel 2.56. Aus wohlfundierter Induktion folgen alle anderen uns bekannten Induktionsprinzipien:

1. $R := \{(n+1, n) \mid n \in \mathbb{N}\}$ ist wohlfundiert. Das ergibt die *„normale" Induktion* über natürliche Zahlen:

$$\forall y.\, (x \,\mathrm{R}\, y \Rightarrow P(y)) \iff \begin{cases} \top \text{ (true)} & \text{falls } x = 0 \\ P(n) & \text{falls } x = n+1 \end{cases}$$

   – d.h. im Induktions-„Schritt" ist für $x = 0$ gerade $P(0)$ zu zeigen (Induktionsanfang), und für $x = n+1$ gerade $P(n) \implies P(n+1)$ (Induktionsschritt im üblichen Sinn).

2. *Course-of-Values-Induktion:* Die Ordnung $>$ auf den natürlichen Zahlen ist wohlfundiert. Wohlfundierte Induktion über $>$ ist genau Course-of-Values-Induktion: wenn $\forall n.\, (\forall k < n.\, P(k)) \Rightarrow P(n)$, dann gilt $P(n)$ für alle n.

3. Wir definieren eine Relation $R \subseteq T_\Sigma(V) \times T_\Sigma(V)$ auf Termen durch

$$R = \{(f(t_1, \ldots, t_n), t_i) \mid f/n \in \Sigma \text{ und } t_1, \ldots, t_n \in T_\Sigma(V) \text{ und } i \in \{1, \ldots, n\}\};$$

   d.h. tRs genau dann, wenn s ein unmittelbarer Unterterm von t ist. Die Relation R ist wohlfundiert; wohlfundierte Induktion über R ist *strukturelle Induktion* über Terme: wenn für jede Operation $f/n \in \Sigma$ aus $P(t_1) \wedge \cdots \wedge P(t_n)$ stets $P(f(t_1, \ldots, t_n))$ folgt, dann gilt $P(t)$ für alle Terme t.

Wir dürfen die Induktionsannahme statt nur für y mit xRy stets auch für y mit xR^+y verwenden (aber natürlich nicht für y mit xR^*y, da wir dann ja die Induktionsbehauptung schon für x selbst annehmen würden); der Grund dafür ist folgendes Lemma.

Lemma 2.57. *Wenn R wohlfundiert ist, dann auch $\Rightarrow R^+$.*

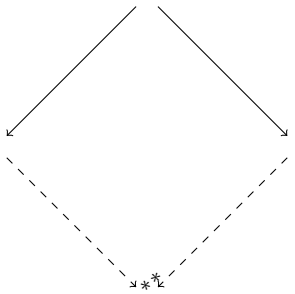
Beweis. Angenommen, es gäbe eine unendliche R^+-Kette

$$\underbrace{x_0 \, R^+ \, x_1}_{x_0 \overset{R}{\to} \tilde{x}_1 \overset{R}{\to} \tilde{x}_2 \overset{R}{\to} \ldots \overset{R}{\to} x_1} \quad R^+ \, x_2 \, R^+ \ldots$$

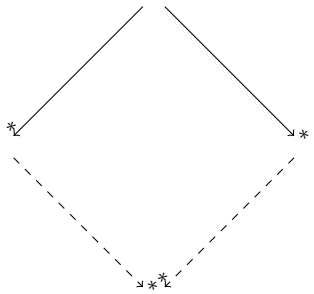
Dann gäbe es auch eine unendliche R-Kette, Widerspruch. □

Beweis von Newmans Lemma (Satz 2.45). (D.h. T stark normalisierend und lokal konfluent $\Rightarrow T$ konfluent.)

Erinnerung:

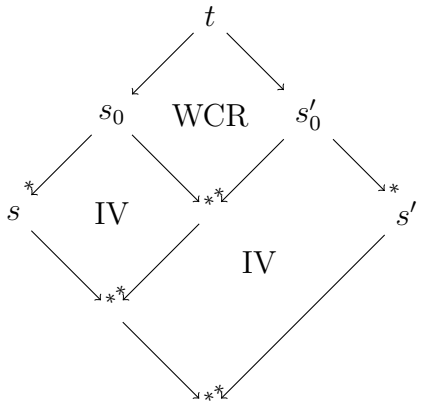


lokale Konfluenz



Konfluenz

Sei also $t \overset{*}{\to} s, s'$, zu zeigen ist dann, dass s, s' zusammenführbar sind. O.E. sind dabei s, t, s' paarweise verschieden, d.h. es gilt $t \to^+ s, s'$. Somit haben wir s_0, s_0' mit $t \to s_0 \to^* s$ und $t \to s_0' \to^* s'$. Wir verwenden wohlfundierte Induktion (bezüglich $\to$, das per starker Normalisierung wohlfundiert ist) über t:



□

3 Der λ-Kalkül (Church/Kleene)

- Lambdakalkül ist funktionales Programmieren mit höheren Funktionen:

```
1 (twice f) x = f(fx)
2
3 map: (a -> b) -> (List a -> List b)
4         map f [] = []
5         map f (x::xs) = (fx)::(map f xs)
```

- Ungetypter Lambdakalkül = Lisp.
- Slogan: λ-Kalkül = der Kalkül der anonymen Funktionen

3.1 Der ungetypte λ-Kalkül

In seiner ursprünglichen Form ist der λ-Kalkül ungetypt und implementiert das Prinzip „Alles ist eine Funktion“. Insbesondere kann alles auf alles angewendet werden. Hierzu hat man eine binäre Operation „Anwendung“ oder „Applikation“, geschrieben als unsichtbare Infixoperation $\_\ \_$, d.h. per Juxtaposition. Der Term $f\ x$ bezeichnet also das Ergebnis der Anwendung von f auf x. Funktionen mit zwei Argumenten emuliert man dann durch zweimalige Anwendung: $(f\ x)\ y$ bezeichnet das Ergebnis der Anwendung von $f\ x$ auf y, wodurch f effektiv zu einer zweistelligen Funktion wird, die man auf Argumente x, y anwendet. Applikation wird daher linksassoziativ geschrieben, d.h. $f\ x\ y$ steht für $(f\ x)\ y$.

Die einzige andere Operation des λ-Kalküls ist die λ-*Abstraktion*, ein Konstrukt für anonyme Funktionen: Wenn t ein Term ist, dann bezeichnet.

$$\lambda x.\, t$$

die anonyme Funktion, die x auf t abbildet. Hierbei kann t von x abhängen (indem es x explizit enthält).

Beispiel 3.1.
- $\lambda x.\, 3 + x$ ist „die Funktion, die zu ihrer Eingabe 3 addiert“ (bisher kommt allerdings + in unserem System noch nicht vor, so dass dies strenggenommen kein legaler Term ist).
- $\lambda x.\, xx$ ist die Funktion, die ihre Eingabe auf sich selbst anwendet.
- $f = \lambda x.\, \lambda y.\, x$ bildet die Eingabe x ab auf die konstante Funktion mit Wert x. Wir können f wie oben angedeutet auch als eine zweistellige Funktion verstehen, die in diesem Fall ein Paar (x, y) von Argumenten auf die erste Komponente x abbildet.

Wir unterstellen ferner weiterhin einen Vorrat V an Variablen; damit sind λ-*Terme* $t \in T(V)$ zusammenfassend definiert durch die Grammatik

$$t ::= x \mid t_1\ t_2 \mid \lambda x.\, t \qquad (x \in V).$$

Wir haben wieder einen Begriff von *Kontext*, hier formal definiert durch die Grammatik

$$C(\cdot) ::= (\cdot) \mid t\ C(\cdot) \mid C(\cdot)\ t \mid \lambda x.\, C(\cdot).$$

Wir verwenden die im vorigen Kapitel eingeführten Begriffe für Relationen auf Termen (stabil, kontextabgeschlossen etc.) weiter. Eine *Kongruenz* ist eine kontextabgeschlossene Äquivalenzrelation.

Notation 3.2. Wir verwenden folgende Konventionen:

- Als Kurzform für mehrere aufeinanderfolgende λ-Abstraktionen schreiben wir $\lambda x_1 \dots x_n . t = \lambda x_1 . \dots . \lambda x_n . t$.
- Der Scope eines λ reicht so weit nach rechts wie möglich: $\lambda x.xx = \lambda x.(xx) \neq (\lambda x.x)x$.

Definition 3.3 (Freie und gebundene Variablen)**.** Sei t ein Term. Dann ist die *Menge der freien Variablen* $FV(t)$ in t rekursiv definiert durch

- $FV(x) = \{x\}$ (wobei x Variable ist.)
- $FV(ts) = FV(t) \cup FV(s)$
- $FV(\lambda x.\, t) = FV(t) \setminus \{x\}$

Eine Variable x heißt *frei* in einem Term t, wenn $x \in FV(t)$. Eine Variable, die in t vorkommt, aber nicht frei ist (z.B. x in $\lambda x.\, s$), heißt *gebunden*.

Definition 3.4 (Substitution)**.** Eine Substitution ist (im wesentlichen wie bisher) eine Abbildung $\sigma : V \to T(V)$, die Variablen auf Terme abbildet. Die Anwendung einer Substitution σ auf Terme ist rekursiv definiert durch

- $x\sigma = \sigma(x)$
- $(ts)\sigma = (t\sigma)(s\sigma)$
- $(\lambda x.t)\sigma = \lambda x.(t\sigma')$, wobei $\sigma' = \sigma[x \mapsto x]$ (d.h. σ' bildet x auf x ab und verhält sich ansonsten wie σ), *wenn* $x \notin FV(\sigma(y))$ für alle $y \in FV(t)$.

Bemerkung 3.5. Die Bedingung $x \notin FV(\sigma(y))$ in der letzten Klausel dient zur Vermeidung eines *Variableneinfangs* (*variable capture*), d.h. der Substitution einer vorher freien Variablen an eine Stelle, an der sie dann gebunden würde. Z.B. sollte vernünftigerweise *nicht* $\lambda x.\, x$ durch Substitution von x für y in $\lambda x.\, y$ entstehen – d.h. die Identitätsfunktion sollte keine Substitutionsinstanz von „konstante Funktion mit Wert y“ sein. Trotzdem wollen wir natürlich jede Substitution auf jeden Term anwenden; wir sehen gleich, wie wir die Substitution $[x/y]$ korrekt auf $\lambda x.\, y$ anwenden.

Definition 3.6. Zwei Terme t_1, t_2 heißen *α-äquivalent* ($t_1 =_\alpha t_2$), wenn sie durch Umbenennung gebundener Variablen auseinander hervorgehen. Formal: $=_\alpha$ ist die von

$$\lambda x.t =_\alpha \lambda y.t[y/x] \qquad \text{wenn } y \notin FV(t).$$

erzeugte Kongruenz ($=_\alpha$ ist offenbar stabil).

Z.B. gilt $\lambda x.\, xy =_\alpha \lambda z.\, zy$, aber nicht $\lambda x.\, y \neq_\alpha \lambda y.\, y$.

Mittels α-äquivalenter Umformung lässt sich der Variableneinfang bei der Substitution umschiffen:

$$(\lambda x.\, y)[x/y] = (\lambda x'.\, y)[x/y] = \lambda x'.\, x$$

3.1.1 β-Reduktion

Der λ-Kalkül ist in erster Linie als Berechnungsmodell konzipiert und hat daher eine Ausführungsvorschrift, die *β-Reduktion*, die das Ausrechnen einer Funktionsanwendung modelliert, z.B. in

$$(\lambda x.\, 3 + x)5 \to_\beta 3 + 5.$$

Das λ-Kalkül ist also im Wesentlichen ein Termersetzungssystem (modulo α-Äquivalenz), mit (in der Basisversion) nur einer Grundreduktion

$$(\beta) \qquad (\lambda x.\, t)x \to_0 t.$$

Man beachte, dass wir dann als Einschrittreduktion $\to_\beta$

$$C((\lambda x.\, t)s) \to_\beta C(t[s/x])$$

bekommen; der Teilterm $(\lambda x.\, t)s$ der linken Seite heißt hierbei ein *β-Redex*. (Es können in anderen Varianten zusätzliche Grundreduktionen dazukommen, insbesondere *η-Reduktion* $\lambda x.\, yx \to_\eta y$.)

Beispiel 3.7.

- $(\lambda x.\, xx)(yx) \to_\beta (yx)(yx) = yx(yx)$
- Nichtterminierung: wir schreiben $\Omega = \lambda x.\, xx$. Dann haben wir

$$\Omega\Omega = (\lambda x.\, xx)\Omega \to_\beta \Omega\Omega \to_\beta \dots$$

- Booleans: $\mathit{true} := \lambda xy.\, x$, $\mathit{false} := \lambda xy.\, y$
- Paare: Wir setzen

$$\begin{aligned} \mathit{fst} &:= \lambda p.\, p\ \mathit{true} \\ \mathit{snd} &:= \lambda p.\, p\ \mathit{false} \\ \mathit{pair} &:= \lambda xy.\, \lambda z.\, zxy \end{aligned}$$

  Dann haben wir z.B. folgende β-Reduktionen:

$$\begin{aligned} \mathit{fst}\ (\mathit{pair}\ x\ y) &= \mathit{fst}\ \big((\lambda xy.\, \lambda z.\, z\ x\ y)\ xy\big) \\ &\to_\beta \mathit{fst}\ ((\lambda y.\, \lambda z.\, z\ x\ y)\ y) \\ &\to_\beta \mathit{fst}\ (\lambda z.\, z\ x\ y) \\ &= (\lambda p.\, p\ \mathit{true})\ (\lambda z.\, z\ x\ y) \\ &\to_\beta (\lambda z.\, z\ x\ y)\ \mathit{true} \\ &\to_\beta \mathit{true}\ x\ y = (\lambda xy.\, x)\ x\ y \\ &\to_\beta (\lambda y.\, x)\ y \to_\beta x \end{aligned}$$

3.1.2 Rekursion

Rekursion bezeichnet allgemein die Definition von Objekten durch Fixpunktgleichungen. Z.B. können wir im λ-Kalkül die übliche rekursive Definition der Fakultätsfunktion

$$\begin{aligned} \mathit{fact} &= \lambda n.\ \texttt{if n=0 then 1 else}\ n \cdot \mathit{fact}(n-1) \\ &=: F\ \mathit{fact} \end{aligned}$$

als Fixpunktgleichung $\mathit{fact} = F\ \mathit{fact}$ mit $F\ f = \lambda n.\ \texttt{if n=0 then 1 else}\ n \cdot (f(n-1))$ verstehen. F ist hierbei „moralisch“ ein *Funktional*, also eine Funktion des Typs

$$F : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$$

(in Wirklichkeit ist das System weiterhin ungetypt), d.h. F erwartet eine Funktion als Argument und gibt dann eine Funktion zurück. Wenn wir einen *Fixpunktkombinator fix* unterstellen, so dass für jedes solche Funktional F

$$\mathit{fix}\ F = F\ (\mathit{fix}\ F)$$

gilt, dann können wir die Definition von *fact* zu

$$\mathit{fact} = \mathit{fix}\ F$$

mit F wie oben umschreiben. In der Tat hat der λ-Kalkül einen solchen Fixpunktkombinator:

Satz 3.8. *Im ungetypten λ-Kalkül*

1. *hat jeder Term t einen* Fixpunkt*, d.h. einen Term s mit $s \to_\beta t\ s$.*
2. *gibt es einen* Fixpunktkombinator *Y, d.h. $Y\ t \to_\beta t\ (Y\ t)$ für alle Terme t.*

Beweis. 1): Setze $W = \lambda x.\, t\ (x\ x)$, $s = W\ W$. Dann

$$s = W\ W = (\lambda x.\, t\ (x\ x))\ W \to_\beta t\ (W\ W) = t\ s.$$

2): Nach 1. tut's $Y = \lambda f.\, (\lambda x.\, f\ (x\ x))\ \lambda x.\, f\ (x\ x)$. □

Korollar 3.9. *Für jeden Term $s[f, x]$ mit freien Variablen f, x existiert ein Term t mit $t\ x \to_\beta^* s[t, x]$.*

M.a.W. wir können rekursive Definitionen der Form $f\ x = s[f, x]$ schreiben, die rechts sowohl die rekursiv definierte Funktion f als auch ihr Argument x erwähnen.

Beweis. Setze $t = Y\ (\lambda f.\, \lambda x.\, s[f, x])$. Dann gilt

$$t\ x \to_\beta (\lambda f.\, \lambda x.\, s[f, x])\ t\ x \to_\beta (\lambda x.\, s[t, x])\ x \to_\beta s[t, x]$$

□

3.1.3 Auswertungsstrategien

Beispiel 3.10. Ob der Ausdruck

$$(\lambda xy.\, x)X(\Omega\Omega)$$

terminiert, hängt von der *Auswertungsstrategie* ab: wenn man zuerst das Funktionsargument $\Omega\Omega$ reduziert (wie dies z.B. ML tun würde), divergiert er (da, wie oben gesehen, $\Omega\Omega$ divergiert), während er terminiert, wenn man (wie etwa Haskell) zuerst die äußeren β-Redexe, also die Anwendung von $(\lambda xy.\, x)$ auf zwei Argumente, reduziert.

Definition 3.11. Die *applikative (auch: leftmost-innermost) Reduktion* $\to_a$ ist induktiv definiert durch:

- $(\lambda x.\, t)s \to_a t[s/x]$, wenn t und s normal sind.
- $\lambda x.\, t \to_a \lambda x.\, t'$, wenn $t \to_a t'$.
- $ts \to_a t's$, wenn $t \to_a t'$.
- $ts \to_a ts'$, wenn $s \to_a s'$ und t normal ist.

Definition 3.12. Die *normale (auch: leftmost-outermost) Reduktion* $\to_n$ ist definiert durch

- $(\lambda x.\, t)s \to_n t[s/x]$.
- $\lambda x.\, t \to_n \lambda x.\, t'$, wenn $t \to_n t'$.
- $ts \to_n t's$, wenn $t \to_n t'$ und t keine λ-Abstraktion ist.
- $ts \to_n ts'$, wenn $s \to_n s'$ und t normal und keine λ-Abstraktion ist.

Bemerkung 3.13. Die applikative Reduktion führt im Beispiel 3.10 zu Nichttermination, während die normale (leftmost-outermost) Reduktion terminiert.

Definition 3.14. Eine Reduktion $t \to_\beta^* s$ heißt *erfolgreich*, wenn sie in einer Normalform s endet.

Satz 3.15 (Standardisierung)**.** *Jede erfolgreiche Reduktion kann durch eine normale Reduktion ersetzt werden. In Formeln:*
$t \to_\beta^* s$, *s Normalform* $\Rightarrow t \to_n^* s$

Beweis. Beweisskizze des Beweises von Barendregt, Felleisen, Richter.

1. Wir definieren die *En-Gros-Reduktion* $\to_g$ induktiv durch

   (a) $x \to_g x$
   (b) $\lambda x.\, t \to_g \lambda x.\, s$, wenn $t \to_g s$
   (c) $ts \to_g t's'$, wenn $t \to_g t'$ und $s \to_g s'$
   (d) $(\lambda x.\, t)s \to_g t'[s'/x]$, wenn $t \to_g t'$ und $s \to_g s'$

Lemma 3.16. *Die transitive und reflexive Hülle der en-Gros-Reduktion ist gleich der der β-Reduktion: $\to_g^* = \to_\beta^*$.*

Lemma 3.17. *Wenn man einen Term t in einem Schrit en-Gros-reduzieren kann auf eine Normalform s, dann kann man diese en-Gros-Reduktion mit einer normalen Reduktion nachstellen, d.h. es gilt: $t \to_g s$, s Normalform $\implies t \to_n^* s$.*

2. *Weak head reduction $\to_w$*:

   Wir können jeden Term t in der Form $t = t_1 \dots t_n$ schreiben, so dass t_1 entweder Variable oder λ (aber keine Applikation) ist.

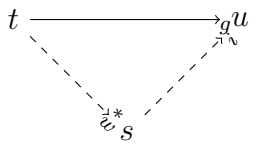
   Dann arbeitet Weak head reduction wie folgt:

   $$(\lambda x.\, t) s_1 s_2 \dots s_n \to_w t[s_1/x] s_2 \dots s_n.$$

3. *Schwache interne Reduktion $\to_i$*: $t_1 \dots t_n \to_i t_1' \dots t_n'$, wenn

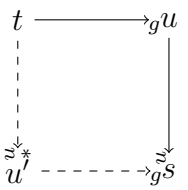
   (a) $t_i \to_g t_i'$ für $i \in \{1, \dots, n\}$ und
   (b) t_1 Variable oder λ-Abstraktion ist.

Lemma 3.18. *Wenn $t \to_g u$, dann existiert ein Term s mit $t \to_w^* s$ und $s \to_i u$:*



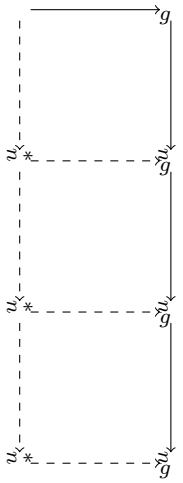
Lemma 3.19.

1. *Wenn $t \to_g u \to_n u$, dann existiert u' mit $t \to_n^* u' \to_g s$:*



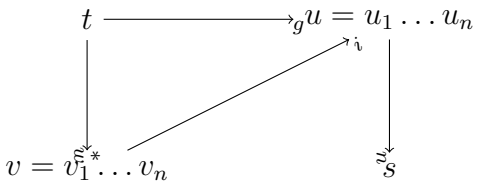
2. *Dasselbe gilt, wenn man $u \to_n s$ durch $u \to_n^* s$ ersetzt.*

Beweis. 2. aus 1. per Induktion:



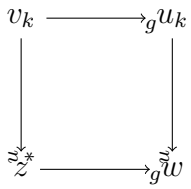
1. Induktion über u.

Per Lemma 3.18 haben wir



mit v_1 (i) Variable oder (ii) λ-Abstraktion. Wir führen nur Fall (i) durch:

Seien $u_1, \ldots, u_{k-1}$ Normalformen, u_k noch nicht; wir reduzieren also in normaler Reduktion u_k, also $s = u_1 \ldots u_{k-1} w u_{k+1} \ldots u_n$ mit $u_k \to_n w$. Per IV haben wir



per Lemma 3.17: $v_i \to_n^* u_i$, $i \in \{1, \ldots, k-1\}$

Weiterhin folgt $t \to_n t'$ aus $t \to_w t'$, so dass auch $t \to_n^* v$ aus $t \to_w^* v$ folgt. Es bleibt also zu zeigen, dass es ein u' gibt, so dass $v \to_n^* u' \to_g s$. Dies ist der Fall:

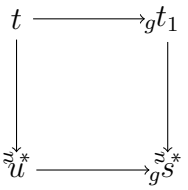
$$\begin{aligned} v = x v_2 \ldots v_{k-1} v_k v_{k+1} \ldots v_n &\to_n^* \\ x u_2 \ldots u_{k-1} z v_{k+1} \ldots v_n &\to_g \\ x u_2 \ldots u_{k-1} w u_{k+1} \ldots u_n &= s \end{aligned}$$

□

Damit kann nun der Beweis der Standardisierung geführt werden:
Sei $t \to_\beta^* s$, s Normalform.
per Lemma 3.16: $t = t_0 \to_g \ldots \to_g t_n \to_g s$
Induktion über n:

- falls $n = 0$ ✓ per Lemma 3.17.
- falls $n > 0$: per Induktionsvoraussetzung gilt:

per Lemma 3.19.2.: $\exists$



per Lemma 3.17: $u \to_n^* s$.

□

Bemerkung 3.20. Warum ist es manchmal dennoch sinnvoll, die applikative Reduktion anzuwenden? Betrachte:
$(\lambda x.\, fxx)((\lambda xy.\, xy)ts)$
Hier verdoppelt sich der hintere Term bei normaler Reduktion anfangs, und man hat gegebenenfalls sehr große Ausdrücke mitzuführen.

3.2 Der einfach getypte λ-Kalkül ($\lambda \to$)

Wir wollen die Termbildung nun stärker kontrollieren (mit dem Ziel, auch bessere Eigenschaften zu bekommen), und lassen nur noch Terme zu, denen wir Typen $\alpha, \beta, \ldots$ zuweisen können. Für den *Typ* der Funktionen von α nach β schreiben wir $\alpha \to \beta$. Z.B. bekommen wir dann

$$\lambda x.\, x : a \to a$$

wobei a eine sogenannte Typvariable ist, für die beliebige Typen eingesetzt werden können. Formal stellt sich dies wie folgt dar.

Definition 3.21. Sei $\mathbf{V}$ eine Menge von *Typvariablen* a, b etc. und $\mathbf{B}$ eine Menge von *Basistypen*, etwa $\mathbf{bool}, \mathbf{int}$. Die Grammatik für *Typen* $\alpha, \beta, \ldots$ ist dann

$$\alpha, \beta ::= a \mid \mathbf{b} \mid \alpha \to \beta \qquad (\mathbf{b} \in \mathbf{B}, \mathbf{a} \in \mathbf{V}).$$

Bemerkung 3.22. Funktionen mit mehreren Argumenten stellen wir wieder mittels *Currying* dar: Der Typ

$$\alpha_1 \to (\alpha_2 \to \ldots \to (\alpha_n \to \beta) \ldots),$$

den wir per Rechtsassoziativität kurz als $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \beta$ schreiben, kann als ein Typ von n-stelligen Funktionen mit Argumenten der Typen $\alpha_1, \ldots, \alpha_n$ angesehen werden.

In der Typisierung nach Church werden abstrahierte Variablen mit Typen annotiert, in der Form $\lambda x : \alpha \,.t$. Man kann in diesem System nur solche Terme hinschreiben, die typisierbar sind. Wir arbeiten hier mit Typisierung nach Curry: wir lassen weiterhin im Prinzip alle Terme zu und sondern dann die Terme aus, die nach unserem System *typisierbar* sind, d.h. für die ein Typ herleitbar ist. Hierzu muss man wissen, welche Typen die in einem Term vorkommenden freien Variablen haben. Ein *Kontext* ist eine Menge

$$\Gamma = \{x_1 : \alpha_1; \ldots; x_n : \alpha_n\}$$

mit paarweise verschiedenen Variablen x_i, denen jeweils ein Typ α_i zugewiesen wird.

Für „im Kontext Γ hat der Term t den Typ α“ schreibt man

$$\Gamma \vdash t : \alpha$$

Diese Relation ist induktiv definiert durch die folgenden Regeln:

$$(Axiom)\ \frac{}{\Gamma \vdash x : \alpha}\ x : \alpha \in \Gamma$$

$$(\to_e)\ \frac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash s : \alpha}{\Gamma \vdash ts : \beta}$$

$$(\to_i)\ \frac{\overbrace{\Gamma, x : \alpha}^{=\Gamma \cup \{x:\alpha\}} \vdash t : \beta}{\Gamma \vdash \lambda x.\, t : \alpha \to \beta}$$

Beispiel 3.23. Wir wollen $\lambda xy.\, xy$ typisieren. Dazu wird zunächst von unten nach oben ein Beweisbaum mit „Lücken“ (hier durch ?? markiert) aufgebaut, die dann später gefüllt werden können.

$$\dfrac{\dfrac{\dfrac{x : ??, y : ?? \ \vdash\ x : ?? \to ?? \qquad x : ??, y : ?? \ \vdash\ y : ??}{x : ??, y : ?? \ \vdash\ xy : ??}\,(\to_e)}{x : ?? \ \vdash\ \lambda y.\, xy : ?? \to ??}\,(\to_i)}{\vdash\ \lambda xy.\, xy : ?? \to ??}\,(\to_i)$$

Nun füllen wir diese Lücken von oben nach unten:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : a \to b, y :\ \text{egal}\ \vdash\ x :\ a \to b}\,(\text{Ax}) \qquad \dfrac{}{x :\ \text{egal}, y : a\ \vdash\ y :\ a}\,(\text{Ax})}{x : a \to b, y : a\ \vdash\ xy : b}\,(\to_e)}{x : a \to b\ \vdash\ \lambda y.\, xy : a \to b}\,(\to_i)}{\vdash\ \lambda xy.\, xy : (a \to b) \to a \to b}\,(\to_i)$$

Beispiel 3.24. Nicht jeder Term ist typisierbar! Beispielsweise ist $\lambda x.\, xx$ (also Ω) nicht typisierbar, d.h. es gibt kein Γ und kein α, so dass gilt: $\Gamma \vdash \Omega : \alpha$, d.h. so dass Ω im Kontext Γ vom Typ α ist.

Damit ergeben sich die folgenden (Berechnungs)probleme. N.B. $(x_1 : \alpha_1, \ldots, x_n : \alpha_n) \vdash t : \beta \iff \vdash \lambda x_1 \ldots x_n . t : \alpha_1 \to \ldots \to \alpha_n \to \beta$.

- gilt $\vdash t : \alpha$? (Typüberprüfung, Type check)
- finde (existiert überhaupt ein?) α mit $\vdash t : \alpha$ (Typinferenz)
- finde (existiert überhaupt ein?) t mit $\vdash t : \alpha$ (Type Inhabitation, automatisches Schließen, Programmsynthese)

Beispiel 3.25.

- $a \to a$ ist inhabited ($\lambda x. x$ ist ein solcher „Bewohner“).
- a ist *nicht* inhabited
- $(a \to a) \to a$ ist *nicht* inhabited

3.2.1 Elementare Eigenschaften

Lemma 3.26 (Weakening)**.** *Sei* $\Gamma \vdash t : \alpha$*. Dann gilt* $\Gamma' \vdash t : \alpha$ *für jeden Kontext* $\Gamma' \supseteq \Gamma$*.*

In Worten: Jede Herleitung klappt auch noch mit zusätzlichen Annahmen.

Beweis. Induktion über Herleitung von $\Gamma \vdash t : \alpha$, mit Fallunterscheidung über die zuletzt angewendete Regel.

1. Letzte Regel (Ax):

$$(Ax)\,\frac{}{\Gamma \vdash x : \alpha} \quad (x : \alpha \in \Gamma)$$

   Wegen $\Gamma' \supseteq \Gamma$ gilt dann auch $x : \alpha \in \Gamma'$, also

$$(Ax)\,\frac{}{\Gamma' \vdash x : \alpha} \quad (x : \alpha \in \Gamma')$$

2. Letzte Regel $(\to_e)$:

$$(\to_e)\,\frac{\begin{array}{c}\vdots\\ \Gamma \vdash t : \beta \to \alpha\end{array} \quad \begin{array}{c}\vdots\\ \Gamma \vdash s : \beta\end{array}}{\Gamma \vdash ts : \alpha}$$

   Nach Induktionsvoraussetzung sind dann $\Gamma' \vdash t : \beta \to \alpha$ und $\Gamma' \vdash s : \beta$ herleitbar, und damit per $(\to_e)$ auch $\Gamma' \vdash ts : \alpha$.

3. Letzte Regel $(\to_i)$:

$$(\to_i)\ \frac{\begin{array}{c}\vdots\\ \Gamma, x:\alpha \vdash t:\beta\end{array}}{\Gamma \vdash \lambda x.\, t : \alpha \to \beta}$$

Ggf. nach Umbenennung können wir annehmen, dass x nicht in Γ' erwähnt wird (strenggenommen verlangt das allerdings ein neues Lemma, demzufolge Typisierung stabil unter Umbenennung von Kontextvariablen ist, wieder zu beweisen per Induktion über Herleitungen). Dann ist $\Gamma', x:\alpha$ ein Kontext, und es gilt $\Gamma, x:\alpha \subseteq \Gamma', x:\alpha$. Nach Induktionsvoraussetzung ist daher $\Gamma', x:\alpha \vdash t:\beta$ herleitbar, per $(\to_i)$ also auch $\Gamma' \vdash \lambda x.\, t : \alpha \to \beta$.

□

Lemma 3.27 (Inversionslemma)**.**

1. $\Gamma \vdash x : \alpha \Rightarrow x : \alpha \in \Gamma$
2. $\Gamma \vdash ts : \beta \Rightarrow \exists \alpha$ *mit* $\Gamma \vdash t : \alpha \to \beta$ *und* $\Gamma \vdash s : \alpha$
3. $\Gamma \vdash \lambda x.\, t : \gamma \Rightarrow \gamma = \alpha \to \beta$ *mit* $\Gamma, x:\alpha \vdash t : \beta$

Beweis. Die Regeln sind syntaxgerichtet. □

3.2.2 Typinferenz

Der Typ eines Ausdrucks ist im Allgemeinen nicht eindeutig bestimmt: $\lambda x.\, x$ kann sowohl den Typ $a \to a$, als auch $(a \to a) \to (a \to a)$ haben. Der erstere Typ in diesem Beispiel ist *allgemeiner*, d.h. man erhält aus ihm den zweiten Typ durch Substitution.

Definition 3.28. Wir bezeichnen mit $TV(\alpha)$ die Menge der in einem Typ α vorkommenden Typvariablen, entsprechend $TV(\Gamma)$ für Kontexte Γ. Eine Typsubstitution σ heißt *Lösung* von $\Gamma \vdash t : \alpha$, wenn $\Gamma\sigma \vdash t : \alpha\sigma$ herleitbar ist, und *allgemeinste Lösung* von $\Gamma \vdash t : \alpha$, wenn σ allgemeinste Typsubstitution unter den Lösungen von $\Gamma \vdash t : \alpha$ ist. Der *Prinzipaltyp von* (Γ, t) (oder von t, wenn Γ leer ist) ist eine allgemeinste Lösung von $\Gamma \vdash t : a$, wobei a „frisch“ ist, d.h. $a \notin TV(\Gamma)$. Das Paar (Γ, t) ist *typisierbar*, wenn eine Lösung von $\Gamma \vdash t : a$ existiert.

Algorithmus 3.29 (Algorithmus W nach Hindley/Milner)**.** Wir bauen $PT(\Gamma; t; \alpha)$ auf als die Menge von Typgleichungen, so dass der *most general unifier* $\sigma = mgu(PT(\Gamma; t; \alpha))$ die allgemeinste Lösung von $\Gamma \vdash t : \alpha$ ist, *wenn* denn Lösungen existieren.

1. $PT(\Gamma;\ x;\ \alpha) = \{\alpha \doteq \beta \mid x : \beta \in \Gamma\}$
2. $PT(\Gamma;\ \lambda x.\, t;\ \alpha) = PT((\Gamma, x:a);\ t;\ b) \cup \{a \to b \doteq \alpha\}$, a, b frisch.
3. $PT(\Gamma; ts; \alpha) = PT(\Gamma; t; a \to \alpha) \cup PT(\Gamma; s; a)$, a frisch

Der Prinzipaltyp von (Γ, t) ist dann die Einschränkung $mgu(PT(\Gamma; t; a))|_{TV(\Gamma,a)}$ von $mgu(PT(\Gamma; t; a))$ auf $TV(\Gamma, a)$.

(Der echte Hindley-Milner-Algorithmus arbeitet allerdings mit einer etwas ausdrucksstärkeren Sprache.) Wir verwenden dabei den Begriff der Unifikation von Gleichungs*mengen*: eine Substitution ist ein *Unifikator* einer Menge $\mathcal{E}$ von Gleichungen, wenn sie Unifikator aller Gleichungen in $\mathcal{E}$ ist.

Beispiel 3.30. Wir berechnen den Prinzipaltyp von $(\emptyset, \lambda xy.\, xy)$:

$$\begin{aligned} PT(\emptyset; \lambda xy.\, xy; a) &= PT(x : b;\ \lambda y.\, xy;\ c) \cup \{a \doteq b \to c\} \\ &= PT(x : b, y : d;\ xy;\ e) \cup \{a \doteq b \to c;\ c \doteq d \to e\} \\ &= PT(x : b, y : d;\ x;\ f \to e) \cup PT(x : b, y : d;\ y;\ f) \\ &\qquad \cup \{a \doteq b \to c;\ c \doteq d \to e\} \\ &= \{b \doteq f \to e, d \doteq f, a \doteq b \to c, c \doteq d \to e\} =: \mathcal{E} \end{aligned}$$

Wir erhalten

$$mgu(\mathcal{E}) = [d/f, d \to e/b, (d \to e) \to d \to e/a, d \to e/c],$$

also hat $\lambda xy.\, xy$ den Prinzipaltyp $mgu(\mathcal{E})(a) = (d \to e) \to d \to e$.

Beispiel 3.31. Wir berechnen den Prinzipaltyp von $(x : a, x\lambda z.\, z)$:

$$\begin{aligned} PT(x : a; x\lambda z.\, z;\ b) &= PT(x : a;\ x;\ c \to b) \cup PT(x : a;\ \lambda z.\, z;\ c) \\ &= \{a \doteq c \to b\} \cup PT(x : a, z : d;\ z;\ e) \cup \{c \doteq d \to e\} \\ &= \{a \doteq c \to b, d \doteq e, c \doteq d \to e\} =: \mathcal{E}. \end{aligned}$$

Wir erhalten

$$mgu(\mathcal{E}) = [d/a, d \to d/e, (d \to d) \to b/a],$$

also hat $(x : a, x\lambda z.\, z)$ den Prinzipaltyp $[(d \to d) \to b/a, b/b]$; Einsetzen in $x : a \vdash x\lambda z.\, z : b$ ergibt $x : (d \to d) \to b \vdash x\lambda z.\, z : b$.

Beispiel 3.32. Wir vollziehen anhand des Algorithmus nach, dass $\lambda x.\, xx$ nicht typisierbar ist: Wir haben

$$\begin{aligned} PT(\emptyset; \lambda x.\, xx; a) &= PT(x : b; xx; c) \cup \{a \doteq b \to c\} \\ &= PT(x : b; x; d \to c\} \cup PT(x : b; x; d) \cup \dots \\ &= \{b \doteq d \to c, b \doteq d, \dots\}. \end{aligned}$$

Diese Gleichungsmenge ist nicht unifizierbar: wir brauchen eine Substitution, die $d \to c$ und d gleich macht, was erkennbar nicht geht (der Algorithmus aus GLoIn scheitert am *occurs check*, weil d in $d \to c$ vorkommt).

Bemerkung 3.33. Es gibt aber stärkere Typsysteme, in denen $\lambda x.xx$ typisierbar ist. Z.B. hat das System $\lambda\cap^{-}$ einen zusätzlichen Typkonstruktor

$$\alpha, \beta ::= \dots \mid \alpha \cap \beta$$

Wir lesen $\alpha \cap \beta$, wie die Notation schon andeutet, als *Durchschnittstyp*, was nicht auf seine Unauffälligkeit als Person hindeuten soll, sondern bedeutet, dass er gerade die Terme enthält, die sowohl Typ α als auch Typ β besitzen. Man hat dann zusätzliche Typisierungsregeln

$$(\cap_i)\ \frac{\Gamma \vdash t : \alpha \quad \Gamma \vdash t : \beta}{\Gamma \vdash t : \alpha \cap \beta}$$

$$(\cap_{e1})\ \frac{\Gamma \vdash t : \alpha \cap \beta}{\Gamma \vdash t : \alpha} \qquad (\cap_{e2})\ \frac{\Gamma \vdash t : \alpha \cap \beta}{\Gamma \vdash t : \beta}$$

$$(\leq)\ \frac{\Gamma \vdash t : \alpha}{\Gamma \vdash t : \beta}\ (\alpha \leq \beta),$$

wobei $\leq$ eine wiederum durch (mehr oder weniger erratbare) Regeln induktiv definierte Ordnung auf Typen ist, z.B. hat man $\alpha \cap \beta \leq \alpha$, und $\alpha \leq \beta$ und $\alpha \leq \gamma$ implizieren zusammen $\alpha \leq \beta \cap \gamma$. Z.B. hat man damit folgende Typherleitung für $\lambda x.\, xx$:

$$\dfrac{\dfrac{(\cap_{e2})\ \dfrac{x : a \cap (a \to b) \vdash x : a \cap (a \to b)}{x : a \cap (a \to b) \vdash x : a \to b} \quad (\cap_{e1})\ \dfrac{x : a \cap (a \to b) \vdash x : a \cap (a \to b)}{x : a \cap (a \to b) \vdash x : a}}{(\to_e)\ \dfrac{x : a \cap (a \to b) \vdash xx : b}{\vdash \lambda x.\, xx : (a \cap (a \to b)) \to b}}}{(\to_i)}$$

Interessanterweise kann man zeigen, dass ein Term genau dann in $\lambda\cap^-$ typisierbar ist, wenn er stark normalisierend ist (van Bakel 1990). Das impliziert allerdings auch eine sehr unangenehme Eigenschaft von $\lambda\cap^-$ (welche?).

Wir beweisen schließlich noch die Korrektheit des Hindley-Milner-Algorithmus:

Satz 3.34. *Ein Paar (Γ, t) ist genau dann typisierbar, wenn $PT(\Gamma; t; a)$ (mit $a \notin TV(\Gamma)$) unifizierbar ist; in diesem Falle ist $mgu(PT(\Gamma; t; a))|_{TV(\Gamma,a)}$ ein Prinzipaltyp von (Γ, t).*

Beweis. Wir zeigen allgemeiner, dass $PT(\Gamma; t; \alpha)$ genau dann unifizierbar ist, wenn $\Gamma \vdash t : \alpha$ eine Lösung hat, und dann $mgu(PT(\Gamma; t; \alpha))|_{TV(\Gamma,\alpha)}$ eine allgemeinste Lösung von $\Gamma \vdash t : \alpha$ ist; äquivalenterweise: eine Typsubstitution σ löst $\Gamma\sigma \vdash t : \alpha\sigma$ genau dann, wenn sie zu einem Unifikator von $PT(\Gamma; t; \alpha)$ erweiterbar ist. Wir beweisen die beiden Implikationen jeweils per Induktion über t.

„$\Rightarrow$“: Jeweils unmittelbar nach Induktionsvoraussetzung und Typregel; z.B. für $t = \lambda x.s$: σ sei erweiterbar zu Unifikator σ' von $PT(\Gamma; \lambda x.s; \alpha) = PT((\Gamma, x : a); s; b) \cup \{a \to b \doteq \alpha\}$ für frische a, b. Nach IV gilt dann $\Gamma\sigma', x : \sigma'(a) \vdash t : b\sigma'$, also per $(\to_i)$ $\Gamma\sigma' \vdash t : \sigma'(a) \to \sigma'(b) = \alpha\sigma'$; da $\sigma'|_{TV(\Gamma,\alpha)} = \sigma$, folgt $\Gamma\sigma \vdash t : \alpha\sigma$.

„$\Leftarrow$“:

$t = x$: Nach dem Inversionslemma folgt aus $\Gamma\sigma \vdash x : \alpha\sigma$ $x : \alpha\sigma \in \Gamma\sigma$, dass $x : \beta \in \Gamma$ und $\alpha\sigma = \beta\sigma$ für ein β, d.h. σ ist Unifikator von $\{\beta \doteq \alpha\} = PT(\Gamma; x; \alpha)$.

$t = us$: Nach dem Inversionslemma folgt aus $\Gamma\sigma \vdash us : \alpha\sigma$, dass $\Gamma\sigma \vdash u : \gamma \to \alpha\sigma$ und $\Gamma\sigma \vdash s : \gamma$ für ein γ. Sei a eine frische Typvariable und $\sigma' = \sigma[a \mapsto \gamma]$. Nach IV ist σ' erweiterbar zu Unifikator von $PT(\Gamma \vdash u : a \to \alpha)$ und von $PT(\Gamma \vdash s : a)$, also von $PT(\Gamma \vdash us : \alpha)$.

$t = \lambda x.\, s$: Nach dem Inversionslemma folgt aus $\Gamma\sigma \vdash \lambda x.\, s : \alpha\sigma$, dass $\alpha\sigma = \beta \to \gamma$ und $\Gamma\sigma, x : \beta \vdash s : \gamma$ für geeignete β, γ. Seien a, b frische Typvariablen und $\sigma' = \sigma[a \mapsto \beta, b \mapsto \gamma]$. Dann löst σ'

$$\Gamma, x : a \vdash s : b,$$

ist also nach IV erweiterbar zu Unifikator σ'' von $PT((\Gamma, x : a); s; b)$. Da ferner $(a \to b)\sigma' = \beta \to \gamma = \alpha\sigma'$, ist σ'' Unifikator von $PT(\Gamma; \lambda x.\, s; \alpha)$. □

3.2.3 Subjekt-Reduktion

Wir beweisen nunmehr, dass das Typsystem kompatibel mit β-Reduktion ist; diese Tatsache kann man als eine Art „Typsicherheit“ interpretieren: Ein Ausdruck verliert durch Auswertung nicht seinen Typ. Wir benötigen zwei weitere einfache Eigenschaften:

Lemma 3.35 (Substitutionslemma)**.**

1. $\Gamma \vdash t : \alpha \implies \Gamma\sigma \vdash t : \alpha\sigma$
2. $\Gamma, x : \alpha \vdash t : \beta$ *und* $\Gamma \vdash s : \alpha \implies \Gamma \vdash t[s/x] : \beta$

Beweis.

1. Induktion über Herleitung von $\Gamma \vdash t : \alpha$
2. Induktion über Herleitung von $\Gamma, x : \alpha \vdash t : \beta$

□

Satz 3.36 (Subjektreduktion)**.** *Wenn* $\Gamma \vdash t : \alpha$ *und* $t \to^* s$*, dann auch* $\Gamma \vdash s : \alpha$*.*

⚠ Die Umkehrung gilt nicht, z.B. haben wir $(\lambda xy.\, y)\lambda x.\, xx \to \lambda y.\, y$; die linke Seite hat keinen Typ, die rechte den Typ $a \to a$.

Beweis. O.E. sei $t \to s$. Wir betrachten zunächst den Fall $t \to_\beta s$, d.h. $t = (\lambda x.\, u)v$, $s = [v/x]u$. Nach Inversionslemma folgt aus $\Gamma \vdash (\lambda x.\, u)v : \alpha$, dass $\Gamma \vdash \lambda x.\, u : \beta \to \alpha$ und $\Gamma \vdash v : \beta$ für ein β. Wiederum per Inversionslemma folgt $\Gamma, x : \beta \vdash tu : \alpha$. Nach dem Substitutionslemma folgt $\Gamma \vdash tu[v/x] : \alpha$.

Den allgemeinen Fall zeigen wir dann per Induktion über Kontexte $C(\cdot)$ (dabei ist die bereits durchgeführte Rechnung der Basisfall $C(\cdot) = (\cdot)$), z.B. für Kontexte $C(\cdot)s$: Sei $t \to t'$, so dass $C(t)s \to C(t')s$; sei $\Gamma \vdash C(t)s : \alpha$. Nach Inversionslemma existiert dann β mit $\Gamma \vdash C(t) : \beta \to \alpha$ und $\Gamma \vdash s : \beta$. Nach IV gilt $\Gamma \vdash C(t') : \beta \to \alpha$, und dann per $(\to_i)$ $\Gamma \vdash C(t')s : \alpha$ □

3.2.4 Der Curry-Howard-Isomorphismus

Die *minimale Logik* ist das Implikationsfragment der intuitonistischen propositionalen Logik IPL, mit Syntax

$$\phi, \psi ::= a \mid \phi \to \psi \qquad a \in V$$

Dies ähnelt nicht zufällig der Grammatik für Typen in $\lambda \to$; Slogan: „Types are propositions“. Beweise in minimaler Logik führt man per natürlichem Schließen:

$$\to_E \frac{\phi \to \psi \qquad \phi}{\psi}$$

$$(\to I) \frac{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}}{\phi \to \psi}$$

Im *Sequentenkalkül* werden lokale Annahmen explizit gemacht: ein *Sequent*

$$\Gamma \vdash \phi$$

liest sich „ϕ ist aus (lokalen) Annahmen Γ herleitbar.“ Das Deduktionssystem hierfür besteht aus den Regeln

$$\to_E \frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

$$\to_I \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}$$

$$(\text{Ax}) \frac{}{\Gamma \vdash \phi} \text{, falls } \phi \in \Gamma$$

Satz 3.37. *Der Sequent $\vdash \phi$ ist genau dann herleitbar, wenn der Typ ϕ* inhabited *ist.*

Beweis. „$\Rightarrow$“: Streichen der Terme aus Herleitung von $\vdash t : \phi$ gibt Herleitung von $\vdash \phi$

„$\Leftarrow$“: Zu $\Gamma = \{\phi_1, \ldots, \phi_n\}$ setze $\overline{\Gamma} = \{x_1 : \phi_1, \ldots, x_n : \phi_n\}$. Wir zeigen per Induktion über die Herleitung von $\Gamma \vdash \phi$, dass ein t mit $\overline{\Gamma} \vdash t : \phi$ existiert. Wir unterscheiden wieder über die zuletzt angewendete Regel:

(Ax): hier wurde also $\Gamma \vdash \phi$ direkt hergeleitet, mit $\phi \in \Gamma$. Dann gilt $\phi = \phi_i$ für ein i, und wir können die Typisierungsaussage $\overline{\Gamma} \vdash x_i : \phi_i$ herleiten.

($\to_E$) Hier endet die Herleitung also auf

$$\frac{\begin{array}{c}\vdots \\ \Gamma \vdash \phi \to \psi\end{array} \qquad \begin{array}{c}\vdots \\ \Gamma \vdash \phi\end{array}}{\Gamma \vdash \psi}$$

Nach IV existieren t, s mit $\overline{\Gamma} \vdash t : \phi \to \psi$ und $\overline{\Gamma} \vdash s : \phi$; per Regel $\to_e$ haben wir dann $\overline{\Gamma} \vdash ts : \psi$.

($\to_I$) Hier endet die Herleitung also auf

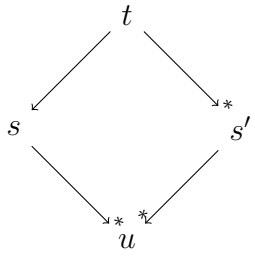
$$\dfrac{\dfrac{\vdots}{\Gamma, \phi \vdash \psi}}{\Gamma \vdash \phi \to \psi}$$

Nach IV existiert t mit $\overline{\Gamma}, x_{n+1} : \phi \vdash t : \psi$. Mit Regel $\to_i$ folgt $\Gamma \vdash \lambda x_{n+1}.\, t : \phi \to \psi$. □

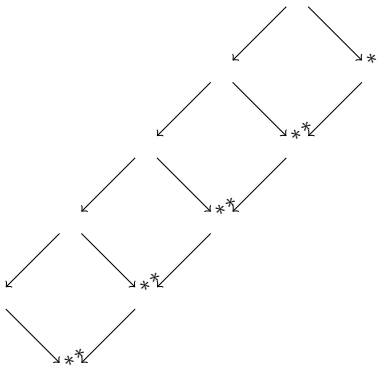
3.2.5 Church-Rosser im λ-Kalkül

Wir zeigen nunmehr, dass β-Reduktion im ungetypten λ-Kalkül (also erst recht in getypten Varianten) konfluent ist. Dabei steht uns Newman's Lemma nicht zur Verfügung, da der ungetypte λ-Kalkül nicht stark normalisierend ist (noch nicht einmal schwach). Wir reduzieren zunächst auf eine marginal einfachere Eigenschaft (formal zwischen Konfluenz und lokaler Konfluenz liegend, in Wirklichkeit aber noch im wesentlichen dasselbe wie Konfluenz):

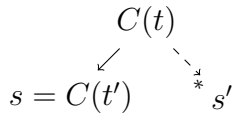
Lemma 3.38 (Streifenlemma)**.** *Wenn $t \to_\beta s$ und $t \to_\beta^* s'$, dann existiert u mit $s \to_\beta^* u$ und $s' \to_\beta^* u$:*



Hieraus folgt unmittelbar Konfluenz; diagrammatisch:



Die Idee zum Beweis des Streifenlemmas ist nun wie folgt. Nach Voraussetzung des Lemmas haben wir die Situation



mit $t \to_0 t'$. Wir verfolgen den β-Redex t durch die rechte Reduktion und reduzieren dann alles, was wir in s' noch von ihm vorfinden. Dazu führen wir einen geeigneten Markierungsmechanismus ein. Wir definieren *markierte Terme* durch die erweiterte Grammatik

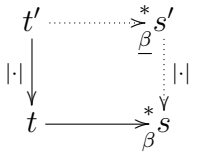
$$t, s := x \mid ts \mid \lambda x.\, t \mid (\underline{\lambda} x.\, t)s,$$

d.h. λ-Abstraktionen in β-Redexen können durch Unterstreichen markiert werden. Wir definieren β-Reduktion $\to_{\underline{\beta}}$ auf markierten Termen wie für λ-Terme, vererben dabei aber natürlich die Markierungen. Auch Substitution wird auf markierten Termen im wesentlichen wie auf normalen Termen definiert. Wir definieren zu einem markierten Term t einen normalen Term $|t|$, indem wir einfach alle Markierungen entfernen (formal per Rekursion, mit einziger interessanter Klausel $|(\underline{\lambda} x.\, t)s| = (\lambda x.\, |t|)|s|)$). Wir definieren außerdem rekursiv eine Funktion ϕ auf markierten Termen, die von innen nach außen vorgehend alle markierten β-Redexe reduziert (und dabei ebenfalls einen normalen Term produziert):

$$\begin{aligned} \phi(x) &= x \\ \phi(ts) &= \phi(t)\phi(s) \\ \phi(\lambda x.\, t) &= \lambda x.\, \phi(t) \\ \phi((\underline{\lambda} x.\, t)s) &= \phi(t)[\phi(s)/x] \end{aligned}$$

In Diagrammen schreiben wir $t \xrightarrow{|\cdot|} |t|$, $t \xrightarrow{\phi} \phi(t)$. Wir halten zunächst fest, dass sich jede Reduktion normaler Terme auch mit Markierungen nachstellen lässt:

Lemma 3.39. *Wenn $t \to_\beta^* s$ und $t' \xrightarrow{|\cdot|} t$, dann existiert s' mit $t' \to_{\underline{\beta}}^* s'$ und $s' \xrightarrow{|\cdot|} s$:*



Beweis. Man reduziert sofort auf den Fall $t \to_\beta s$. Der ist aber klar, weil t' bis auf die Markierungen dieselbe Gestalt hat wie t und daher auch dieselben Reduktionen erlaubt. □

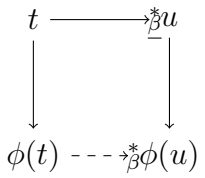
Lemma 3.40. *Seien t, s, u markierte Terme. Dann gilt*

a) Wenn $x \neq y$ und $x \notin FV(u)$, dann

$$t[s/x][u/y] = t[u/y][s[u/y]/x]$$

b) $\phi(t[s/x]) = \phi(t)[\phi(s)/x]$

c) Wenn $t \to_{\underline{\beta}}^* u$, dann $\phi(t) \to_\beta^* \phi(s)$:



Beweis.

a) Induktion über Struktur von t:

$t = x$: dann $x[s/x][u/y] = s[u/y]$ und $x[u/y][s[u/y]/x] = x[s[u/y]/x] = s[u/y]$. (Letzteres verwendet in der ersten Umformung die Annahme $x \neq y$.)

$t = y$: dann $y[s/x][u/y] = u$ (da $x \neq y$) und $y[u/y][s[u/y]/x] = u[s[u/y]/x] = u$ (da $x \neq y$ und $x \notin FV(u)$).

Die restlichen Fälle ($t = z \notin \{x, y\}$, $t = t_1 t_2$, $t = \lambda w.\, t_0$, $t = (\underline{\lambda} x.\, t_0) t_1$) sind leicht.

b) Induktion über Struktur von t. Einziger interessanter Fall ist $t = (\underline{\lambda} y.\, u)\, v$, o.E. mit $y \notin FV(s)$ und $x \neq y$, damit auch $y \notin \phi(s)$; dann haben wir

$$\begin{aligned}
\phi((\underline{\lambda} y.\, u)v)[s/x]) &= \phi((\underline{\lambda} y.\, u[s/x])v[s/x]) && (x \neq y, y \notin FV(s)) \\
&\underset{\text{Def. } \phi}{=} \phi(u[s/x])[\phi(v[s/x])/y] \\
&\underset{2\times\text{IV}}{=} \phi(u)[\phi(s)/x][\phi(v)[\phi(s)/x]/y] \\
&\underset{a)}{=} \phi(u)[\phi(v)/y][\phi(s)/x] && (x \neq y, y \notin FV(\phi(s))) \\
&\underset{\text{Def. } \phi}{=} \phi((\underline{\lambda} y.\, u)v)[\phi(s)/x]
\end{aligned}$$

c) Wieder reduzieren wir sofort auf den Fall $t \to_{\underline{\beta}} s$. Wir betrachten zunächst den Fall $t \to_0 s$; hier unterscheiden wir wiederum danach, ob der β-Redex t markiert ist:

(i) Markierter Fall:

$$\begin{array}{ccc}
(\underline{\lambda} x.\, u)v & \xrightarrow{\quad\underline{\beta}\quad} & u[v/x] \\
{\scriptstyle\phi}\downarrow & & \downarrow{\scriptstyle\phi} \\
\phi(u)[\phi(v)/x] & = & \phi(u)[\phi(v)/x]
\end{array}$$

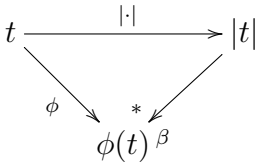
(ii) Unmarkierter Fall:

$$\begin{array}{ccc}
(\lambda x.\, u)v & \xrightarrow{\quad\underline{\beta}\quad} & u[v/x] \\
{\scriptstyle\phi}\downarrow & & \downarrow{\scriptstyle\phi} \\
(\lambda x.\, \phi(u))\phi(v) & \xrightarrow{\quad\beta\quad} & \phi(u)[\phi(v)/x]
\end{array}$$

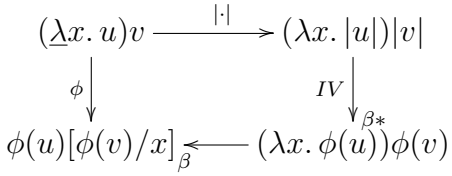
Der allgemeine Fall ist dann eine einfache Induktion über Kontexte: Wenn z.B. die Reduktion $t \to_0 t'$ im Kontext $C(\cdot)s$ stattfindet, haben wir nach IV $\phi(C(t)) \to_\beta^* \phi(C(t'))$ und damit auch $\phi(C(t)s) = \phi(C(t))s \to_\beta^* \phi(C(t'))s = \phi(C(t')s)$.

□

Lemma 3.41. *Es gilt $|t| \to_\beta^* \phi(t)$; als Diagramm:*

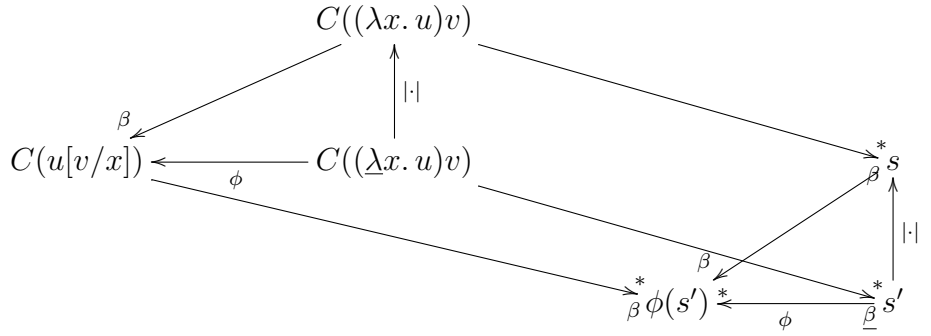


Beweis. Induktion über t, einziger interessanter Fall:



□

Damit ist der Beweis des Streifenlemmas leicht per Diagramm zu führen: Wir markieren den Redex $(\lambda x.\, u)v$; damit erhalten wir das hintere Dreieck, weil in $C((\underline{\lambda} x.\, u)v)$ der einzige markierte Redex eben $(\underline{\lambda} x.\, u)v$ ist. Per Lemma 3.39 bekommen wir s' wie in der hinteren Fläche. Dann haben wir das vordere Dreieck per Lemma 3.41 und die untere Fläche per Lemma 3.40.



Korollar 3.42. *Jeder λ-Term hat höchstens eine NF.*

3.3 Starke Normalisierung für $\lambda\to$

Wir beweisen nunmehr, dass $\lambda\to$ stark normalisierend ist. Wir beweisen dieselbe Eigenschaft später noch einmal für das stärkere Typsystem $\lambda 2$, so dass dieser Abschnitt bei Bedarf übersprungen werden kann.

Die Hauptidee am Beweis ist die Definition einer Semantik für Typen α als Teilmengen

$$[\![\alpha]\!] \subseteq SN := \{t \in \Lambda \mid t \text{ stark normalisierend}\},$$

wobei Λ die Menge aller λ-Terme ist; wenn wir Korrektheit des Typsystems bezüglich dieser Semantik zeigen, d.h wenn wir zeigen, dass jeder typisierbare Term tatsächlich zur Interpretation seines Typs gehört, ist das Resultat offenbar bewiesen. Wir geben die Semantik rekursiv an: Für $A, B \subseteq \Lambda$ schreiben wir

$$A \to B = \{t \in \Lambda \mid \forall s \in A.\, ts \in B\}$$

und setzen dann

$$[\![a]\!] = SN$$
$$[\![\alpha \to \beta]\!] = [\![\alpha]\!] \to [\![\beta]\!].$$

Definition 3.43. Eine Teilmenge $A \subseteq SN$ heißt *saturiert*, wenn

1. $xt_1 \dots t_n \in A$ für alle Variablen x und alle $t_1, \dots, t_n \in SN$, $n \geq 0$.
2. $t[x \mapsto s]u_1 \dots u_n \in A \Rightarrow (\lambda x.\, t)su_1 \dots u_n \in A$ für alle $s \in SN$.

Wir setzen dann

$$SAT = \{A \subseteq \Lambda \mid A \text{ saturiert}\}.$$

Lemma 3.44 (Saturiertheitslemma)**.** $[\![\alpha]\!] \in SAT$ *für alle* α*.*

Beweis. Induktion über α.

1. zu zeigen: $[\![a]\!] = SN \in SAT$

   (a) Sei x eine Variable und $t_1, \dots, t_n \in SN$. Zu zeigen ist $xt_1 \dots t_n \in SN$. Das ist aber klar: Jeder Redukt von $xt_1 \dots t_n$ ist von der Form $xt'_1 \dots t'_n$ mit $t_i \to^*_\beta t'_i$ (also $t'_i \in SN$), so dass man aus einer unendlichen Reduktionssequenz für $xt_1 \dots t_n$ auch eine für eines der t_i gewinnen würde, im Widerspruch zu $t_i \in SN$.

   (b) Sei $s \in SN$ und $t[s/x]u_1 \dots u_n \in SN$; dann gilt $t, u_1, \dots, u'_n in SN$. Zu zeigen ist $v := (\lambda x.\, t)su_1 \dots u_n \in SN$. Da $t, s, u_1, \dots, u_n \in SN$, hat jede unendliche Reduktionssequenz von v die Form

   $$(\lambda x.\, t)su_1 \dots u_n \xrightarrow{*} (\lambda x.\, t')s'u'_1 \dots u'_n \to t'[x \mapsto s']u'_1 \dots u'n \to \dots,$$

   im Widerspruch zu $t[s/x]u_1 \dots u_n \in SN$.

2. Seien $A := [\![\alpha]\!], B := [\![\beta]\!]$ saturiert; zu zeigen ist, dass $A \to B$ saturiert ist.

   (a) $A \to B \subseteq SN$: Sei $t \in A \to B$. Sei x eine Variable. Da A saturiert ist, gilt $x \in A$, somit $tx \in B$ per Definition von $A \to B$, also $tx \in SN$ und somit $t \in SN$.

(b) Seien $r_1, \ldots, r_n \in SN$; zu zeigen ist $xr_1 \ldots r_n \in A \to B$. Sei also $s \in A \subseteq SN$, zu zeigen ist dann $xr_1 \ldots r_n s \in B$. Da $s \in SN$, folgt dies aus Saturiertheit von B.

(c) Sei $s \in SN$ und $t[x \mapsto s]r_1 \ldots r_n \in A \to B$. Zu zeigen ist $(\lambda x.\, t)sr_1 \ldots r_n \in A \to B$. Sei also $v \in A$, zu zeigen ist dann $(\lambda x.\, t)sr_1 \ldots r_n v \in B$. Per Saturiertheit von B reicht dazu $t[x \mapsto s]sr_1 \ldots r_n v \in B$, was aus $v \in a$ und $t[x \mapsto s]r_1 \ldots r_n \in A \to B$ folgt.

□

Definition 3.45 (Erfülltheit/Konsequenz)**.** Sei σ eine Substitution. Dann schreiben wir

$$\begin{aligned}
\sigma \models t : \alpha &:\iff t\sigma \in [\![\alpha]\!] && (\sigma \textit{ erfüllt } t : \alpha)\\
\sigma \models \Gamma &:\iff \forall (x : \alpha) \in \Gamma.\, \sigma \models x : \alpha && (\sigma \textit{ erfüllt } \Gamma)\\
\Gamma \models t : \alpha &:\iff \forall \sigma.\, (\sigma \models \Gamma \Rightarrow \sigma \models (t : \alpha)) && (t : \alpha \text{ ist } \textit{Konsequenz} \text{ von } \Gamma)
\end{aligned}$$

Lemma 3.46 (Korrektheit)**.** *Wenn* $\Gamma \vdash t : \alpha$*, dann* $\Gamma \models t : \alpha$*.*

Diese Aussage hat den gleichen Charakter wie Korrektheitsaussagen über Beweissysteme: Wenn man aus Typisierungsannahmen Γ mittels der Typregeln herleiten kann, dass t Typ α hat, dann ist $t : \alpha$ auch eine Konsequenz aus Γ in der Semantik.

Beweis. Formal führen wir eine Induktion über die Herleitung von $\Gamma \vdash t : \alpha$ durch; informell heißt dies, dass wir zeigen, dass alle Regeln des Typsystems korrekt bezüglich der Semantik sind.

$$(\mathrm{Ax})\ \frac{}{\Gamma \vdash x : \alpha}$$

Hier ist zu zeigen, dass $\Gamma \models x : \alpha$; das gilt trivialerweise.

$$(\to_e)\ \frac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash s : \alpha}{\Gamma \vdash ts : \beta}$$

Sei $\Gamma \models t : \alpha \to \beta$ und $\Gamma \models s : \alpha$. Zu zeigen ist dann $\Gamma \models ts : \beta$. Sei also $\sigma \models \Gamma$. Nach Annahme gilt $\sigma \models t : \alpha \to \beta$ und $\sigma \models s : \alpha$, d.h. $t\sigma \in [\![\alpha]\!] \to [\![\beta]\!]$ und $s\sigma \in [\![\alpha]\!]$, also $(ts)\sigma = (t\sigma)s\sigma \in [\![\beta]\!]$, d.h. $\sigma \models ts : \beta$.

$$(\to_i)\ \frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.\, t \ : \ \alpha \to \beta}$$

Sei $\Gamma, x : \alpha \models t : \beta$ und $\sigma \models \Gamma$. O.E. sei x frisch. Zu zeigen ist $(\lambda x.\, t)\sigma \in [\![\alpha]\!] \to [\![\beta]\!]$. Sei also $v \in [\![\alpha]\!]$, zu zeigen ist dann $((\lambda x.\, t)\sigma)v \in [\![\beta]\!]$. Dazu rechnen wir

$$\begin{aligned}
&((\lambda x.\, t)\sigma)v \in [\![\beta]\!] \\
&\Longleftarrow t\sigma[x \mapsto v] \in [\![\beta]\!] && ([\![\beta]\!] \text{ saturiert}) \\
&\iff \sigma[x \mapsto v] \models t : \beta \\
&\Longleftarrow \sigma[x \mapsto v] \models \Gamma, x : \alpha && (\text{Annahme}) \\
&\iff \sigma[x \mapsto v] \models x : \alpha && (\sigma \models \Gamma) \\
&\iff v \in [\![\alpha]\!];
\end{aligned}$$

letzteres gilt nach Voraussetzung. □

Daraus folgt im wesentlichen sofort unser Zielresultat:

Satz 3.47. *$\lambda\to$ ist stark normalisierend.*

Beweis. Sei $\Gamma \vdash t : \alpha$. Nach dem Korrektheitslemma folgt $\Gamma \models t : \alpha$. Setze $\sigma = [\,]$. Dann gilt $\sigma \models \Gamma$, da $x \in [\![\alpha]\!]$ für alle α per Saturiertheit von $[\![\alpha]\!]$. Es folgt $\sigma \models t : \alpha$, d.h. $t = t\sigma \in [\![\alpha]\!] \subseteq SN$. □

4 Induktive Datentypen

Induktive Datentypen sind Typen endlicher (allgemeiner: wohlfundiert aufgebauter) Objekte, für die wir somit über Rekursions- und Induktionsprinzipien verfügen. Wir beginnen mit einem einfachen und bekannten Beispiel (in ab jetzt standardisierter Syntax):

Beispiel 4.1.

```
1 data Nat where
2         0: () -> Nat
3         Suc: Nat -> Nat
```

- Die obige Datentypdeklaration definiert eine *Signatur* Σ_{Nat}, also eine Menge von *Funktionsymbolen*. Diese werden *Konstruktoren* genannt.
- Die *Semantik* von *Nat* ist definiert als

$$[\![Nat]\!] = T_{\Sigma_{Nat}}(\emptyset) = \{0, Suc(0), Suc(Suc(0)), \dots\}.$$

Wir erinnern an dieser Stelle an einen Begriff aus GLoIn:

Definition 4.2. Ein Σ-Modell $\mathfrak{M}$ besteht aus

- einem *Grundbereich* (oder Träger oder Universum), d.h. einer Menge M

- für jede n-stellige Funktion f, also $f/n \in \Sigma$, einer Abbildung

$$\mathfrak{M}[\![f]\!] : M^n \to M.$$

Im Kontext von induktiven Datentypen (d.h. insbesondere, wenn Σ nur aus Funktionssymbolen besteht) bezeichnen wir Σ-Modelle auch als Σ-*Algebren*.

Beispiel 4.3. Σ_{Nat}-Algebren $\mathfrak{M}$ bestehen demnach aus einer Trägermenge M, einer Konstanten $\mathfrak{M}[\![0]\!] \in M$, und einer einstelligen Funktion $\mathfrak{M}[\![Suc]\!] : M \to M$. Insbesondere wird $[\![Nat]\!]$ eine Σ_{Nat}-Algebra per

$$[\![Nat]\!][\![0]\!] = 0$$
$$[\![Nat]\!][\![Suc]\!](t) = Suc(t).$$

Definition 4.4. Ein Σ-*Homomorphismus* $h : \mathfrak{M} \to \mathfrak{N}$ zwischen Σ-Algebren $\mathfrak{M}$, $\mathfrak{N}$ ist eine Funktion $h : M \to N$ mit

$$h(\mathfrak{M}[\![g]\!](x_1, \ldots, x_n)) = \mathfrak{N}[\![g]\!](h(x_1), \ldots, h(x_n))$$

für alle $g/n \in \Sigma$, $x_1, \ldots, x_n \in M$. Wenn h ferner bijektiv ist, heißt h Σ-*Isomorphismus*. Wenn Σ aus dem Kontext klar ist, wird es in der Notation ausgelassen.

Lemma 4.5. *Sei $\mathfrak{M}$ eine Σ-Algebra. Dann ist $id : \mathfrak{M} \to \mathfrak{M}$ ein Homomorphismus. Wenn $h_1 : \mathfrak{M} \to \mathfrak{N}$ und $h_2 : \mathfrak{N} \to \mathfrak{P}$ Homomorphismen sind, dann ist $h_1 \circ h_2 : \mathfrak{M} \to \mathfrak{P}$ ein Homomorphismus.*

Lemma 4.6. *Wenn $h : \mathfrak{M} \to \mathfrak{N}$ ein Σ-Isomorphismus ist, dann ist $h^{-1} : \mathfrak{N} \to \mathfrak{M}$ ein Isomorphismus.*

Beweis. h^{-1} ist bekanntermaßen ebenfalls bijektiv; es bleibt zu zeigen, dass h^{-1} ein Homomorphismus ist. Seien also $f/n \in \Sigma$, $y_1, \ldots, y_n \in N$. Zu zeigen ist

$$h^{-1}(\mathfrak{N}[\![f]\!](y_1, \ldots, y_n) = \mathfrak{M}[\![f]\!](h^{-1}(y_1), \ldots, h^{-1}(y_n)).$$

Da h injektiv ist, reicht dazu

$$h(h^{-1}(\mathfrak{N}[\![f]\!](y_1, \ldots, y_n))) = h(\mathfrak{M}[\![f]\!](h^{-1}(y_1), \ldots, h^{-1}(y_n))).$$

Die linke Seite dieser Gleichung ist gleich $\mathfrak{N}[\![f]\!](y_1, \ldots, y_n)$; die rechte ist per Homomorphie von h ebenfalls gleich $\mathfrak{N}[\![f]\!](h(h^{-1}(y_1)), \ldots, h(h^{-1}(y_n))) = \mathfrak{N}[\![f]\!](y_1, \ldots, y_n)$. □

Bemerkung 4.7 (Erinnerung: Restklassen)**.** Man schreibt

$$m \equiv n \ (4) \iff 4 \mid (m - n)$$

für ganze Zahlen m, n, und definiert damit eine Äquivalenzrelation auf $\mathbb{Z}$. Dann ist die *Restklasse*

$$[n]_4 = \{m \in \mathbb{Z} \mid m \equiv n \ (4)\}$$

die Äquivalenzklasse von n modulo 4; die Menge der Restklassen modulo 4 bezeichnen wir mit $\mathbb{Z}/4\mathbb{Z}$, also

$$\mathbb{Z}/4\mathbb{Z} = \{[n]_4 \mid n \in \mathbb{Z}\} = \{[0]_4, [1]_4, [2]_4, [3]_4\}.$$

Addition von Restklassen ist definiert durch

$$[n]_4 + [m]_4 = [n+m]_4.$$

Das ist *wohldefiniert*: Wenn $n_1 \equiv n_2\ (4)$ und $m_1 \equiv m_2\ (4)$, dann $n_1 + m_1 \equiv n_2 + m_2\ (4)$

Beispiel 4.8. Damit erhalten wir ein weiteres Beispiel einer Σ_{Nat}-Algebra: Wir definieren eine Σ_{Nat}-Algebra $\mathfrak{N}$ mit Trägermenge

$$N = \mathbb{Z}/4\mathbb{Z}$$

durch

$$\begin{aligned}\mathfrak{N}[\![0]\!] &= [0]\\ \mathfrak{N}[\![Suc]\!][n]_4 &= [n+1]_4.\end{aligned}$$

Mit $\mathfrak{M} := [\![Nat]\!]$ wie oben erhalten wir nun einen Σ_{Nat}-Homomorphismus $h : \mathfrak{M} \to \mathfrak{N}$:

$$\begin{aligned}h &: M \to N\\ Suc^n(0) &\mapsto [n]_4.\end{aligned}$$

Die nachzuprüfenden Eigenschaften sind hier

- $h(\mathfrak{M}[\![0]\!]) = \mathfrak{N}[\![0]\!]$
- $h(\mathfrak{M}[\![Suc]\!](x)) = \mathfrak{N}[\![Suc]\!](h(x))$.

Dies rechnen wir wie folgt nach:

$$\begin{aligned}h(\mathfrak{M}[\![0]\!]) = h(0) = h(Suc^0(0)) = [0]_4 = \mathfrak{N}[\![0]\!]\\ h(\mathfrak{M}[\![Suc]\!](Suc^n(0))) = h(Suc^{n+1}(0)) = [n+1]_4 = \mathfrak{N}[\![Suc]\!]([n]_4) = \mathfrak{N}[\![Suc]\!](h(Suc^n(0))).\end{aligned}$$

Definition 4.9. Eine Σ-Algebra $\mathfrak{M}$ heißt *initial*, wenn für jede Σ-Algebra $\mathfrak{N}$ genau ein Σ-Homomorphismus $\mathfrak{M} \to \mathfrak{N}$ existiert.

Satz 4.10. *$T_\Sigma(\emptyset)$ mit*

$$\mathfrak{M}[\![f]\!](t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$$

ist initiale Σ-Algebra. Für eine Σ-Algebra $\mathfrak{N}$ ist der eindeutige Homomorphismus $h : \mathfrak{M} \to \mathfrak{N}$ gegeben durch

$$h(t) = \mathfrak{N}[\![t]\!].$$

Insbesondere ist also $[\![Nat]\!]$ eine initiale Σ_{Nat}-Algebra.

Beweis. Eine Abbildung $h : M \to N$ ist Σ-Homomorphismus $\mathfrak{M} \to \mathfrak{N}$ genau dann, wenn

$$h(f(t_1, \ldots, t_n)) \overset{\text{Def.}}{=} h(\mathfrak{M}[\![f]\!](t_1, \ldots, t_n)) = \mathfrak{N}[\![f]\!](h(t_1, \ldots, t_n))$$

für $f/n \in \Sigma$, $t_1, \ldots, t_n \in M = T_\Sigma(\emptyset)$. Dies ist genau die rekursive Definition der Auswertung geschlossener Terme in $\mathfrak{N}$. □

Satz 4.11. *Die initiale Σ-Algebra ist eindeutig bis auf Isomorphie.*

Beweis. Seien $\mathfrak{M}, \mathfrak{N}$ initiale Σ-Algebren. Dann haben wir Σ-Homomorphismen

$$id \circlearrowright \mathfrak{M} \underset{g}{\overset{f}{\rightleftarrows}} \mathfrak{N} \circlearrowright id$$

Per Eindeutigkeit gilt dann $g \circ f = id$ und $f \circ g = id$, d.h. f und g sind gegenseitig inverse Bijektionen. □

Definition 4.12 (Notation zu Mengenkonstruktionen)**.** Seien X_1, X_2 Mengen (man stelle sich diese als „Typen“ vor). Dann schreiben wir (wie üblich)

$$\begin{aligned} X_1 \times X_2 &= \{(x_1, x_2) \mid x_i \in X_i \text{ für } i = 1, 2\} && (\text{„struct“}) \\ X_1 + X_2 &= \{(i, x) \mid i = 1, 2 \text{ und } x \in X_i\} && (\text{„union“}) \\ 1 &= \{*\} && (\text{„()“ in Haskell}). \end{aligned}$$

Sei $f_i : X_i \to Y_i$, $g_i : X_i \to Z$ und $h_i : Z \to X_i$ für $i \in \{1, 2\}$. Dann haben wir Abbildungen

$$\begin{aligned} f_1 \times f_2 &: X_1 \times X_2 \to Y_1 \times Y_2, & (f_1 \times f_2)(x_1, x_2) &= (f_1(x_1), f_2(x_2)) \\ f_1 + f_2 &: X_1 + X_2 \to Y_1 + Y_2, & (f_1 + f_2)(i, x) &= (i, f_i(x)) \\ [g_1, g_2] &: X_1 + X_2 \to Z, & [g_1, g_2](i, x) &= g_i(x) \\ \langle h_1, h_2 \rangle &: Z \to X_1 \times X_2, & \langle h_1, h_2 \rangle(z) &= (h_1(z), h_2(z)) \\ in_i &: X_i \to X_1 + X_2, & in_i(x) &= (i, x) \\ \pi_i &: X_1 \times X_2 \to X_i, & \pi_i(x_1, x_2) &= x_i \\ 1 &: 1 \to 1 \end{aligned}$$

Lemma 4.13. *In Bezeichnungen wie oben gilt*

- $[g_1, g_2] \circ in_i = g_i$
- $f_1 + f_2 = [in_1 \circ f_1, in_2 \circ f_2]$
- $[r \circ in_1, r \circ in_2] = r$ *für* $r : X_1 + X_2 \to Z$.

- $\pi_i \circ \langle h_1, h_2 \rangle = h_i$
- $f_1 \times f_2 = \langle f_1 \circ \pi_1, f_2 \circ \pi_2 \rangle$
- $\langle \pi_1 \circ f, \pi_2 \circ f \rangle = f$ *für* $f : Z \to X_1 \times X_2$

Insbesondere ist jede Abbildung $Z \to X_1 \times X_2$ von der Form $\langle h_1, h_2 \rangle$, und jede Abbildung $X_1 + X_2 \to Z$ von der Form $[g_1, g_2]$.

Beweis. Wir haben

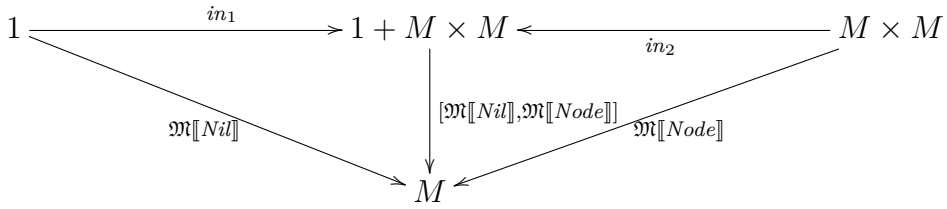
$$
\begin{aligned}
&[f_1, f_2](in_i(x)) = [f_1, f_2](i, x) = f_i(x) \\
&[in_1 \circ f_1, in_2 \circ f_2](i, x) = in_i(f_i(x)) = (i, f_i(x)) = (f_1 + f_2)(i, x) \\
&[r \circ in_1, r \circ in_2](i, x) = r(in_i(x)) = r(i, x) \\
&\pi_i \circ \langle h_1, h_2 \rangle(z) = \pi_i(h_1(z), h_2(z)) = h_i(z) \\
&\langle f_1 \circ \pi_1, f_2 \circ \pi_2(x_1, x_2) = (f_1(\pi_1(x_1, x_2)), f_2(\pi_2(x_1, x_2))) = \\
&\qquad\qquad (f_1(x_1), f_2(x_2)) = \langle f_1, f_2 \rangle(x_1, x_2) \\
&\langle \pi_1 \circ f, \pi_2 \circ f \rangle(z) = (\pi_1(f(z)), \pi_2(f(z))) = f(z).
\end{aligned}
$$

□

Beispiel 4.14. Wir betrachten die Datentypdeklaration

```
1 data Tree where
2         Nil: () -> Tree
3         Node: Tree * Tree -> Tree
```

Eine Σ_{tree}-Algebra $\mathfrak{M}$ lässt sich dann darstellen als eine Abbildung $\alpha = [\mathfrak{M}[\![Nil]\!], \mathfrak{M}[\![Node]\!]] : 1 + M \times M \to M$, und umgekehrt induziert jede Abbildung $\alpha : 1 + M \times M \to M$ eine Σ_{tree}-Algebra per $\mathfrak{M}[\![Nil]\!] = \alpha(in_1(*))$ und $\mathfrak{M}[\![Node]\!](x, y) = \alpha(in_2(x, y))$.



Eine Abbildung $f : M \to N$ erfüllt genau dann die Homomorphiebedingung für *Node*, wenn das Diagramm

$$
\begin{array}{ccc}
M \times M & \xrightarrow{f \times f} & N \times N \\
{\scriptstyle \mathfrak{M}[\![Node]\!]}\downarrow & & \downarrow{\scriptstyle \mathfrak{N}[\![Node]\!]} \\
M & \xrightarrow[f]{} & N
\end{array}
\tag{7}
$$

kommutiert (das heißt nämlich gerade, dass $f(\mathfrak{M}[\![Node]\!](x,y)) = \mathfrak{N}[\![Node]\!]((f\times f)(x,y)) = \mathfrak{N}[\![Node]\!](f(x),f(y))$). Ebenso erfüllt f die Homomorphiebedingung bezüglich Nil genau dann, wenn das Diagramm

$$\begin{array}{ccc} 1 & \xrightarrow{1} & 1 \\ {\scriptstyle \mathfrak{M}[\![Nil]\!]}\downarrow & & \downarrow{\scriptstyle \mathfrak{N}[\![Nil]\!]} \\ M & \xrightarrow[f]{} & N \end{array}$$

kommutiert (das heißt nämlich, dass $f(\mathfrak{M}[\![Nil]\!]()) = \mathfrak{N}[\![Nil]\!]()$).

Diese beiden Diagramme können wir in einem Diagramm zusammenfassen: f ist Σ_{Tree}-Homomorphismus genau dann, wenn das Diagramm

$$\begin{array}{ccc} 1 + M\times M & \xrightarrow{1+f\times f} & 1 + N\times N \\ {\scriptstyle [\mathfrak{M}[\![Nil]\!],\mathfrak{M}[\![Node]\!]]}\downarrow & & \downarrow{\scriptstyle [\mathfrak{N}[\![Nil]\!],\mathfrak{M}[\![Node]\!]]} \\ M & \xrightarrow[f]{} & N \end{array} \tag{8}$$

kommutiert: z.B. erhalten wir aus diesem Diagramm das Diagramm (7) durch Postkomponieren mit in_2:

$$\begin{aligned} f\circ\mathfrak{M}[\![Node]\!] &= f\circ[\mathfrak{M}[\![Nil]\!],\mathfrak{M}[\![Node]\!]]\circ in_2 && \text{(Lemma 4.13)}\\ &= [\mathfrak{N}[\![Nil]\!],\mathfrak{M}[\![Node]\!]]\circ(1+f\times f)\circ in_2 && (8)\\ &= [\mathfrak{N}[\![Nil]\!],\mathfrak{M}[\![Node]\!]]\circ[\ldots, in_2\circ(f\times f)]\circ in_2 && \text{(Lemma 4.13)}\\ &= [\mathfrak{N}[\![Nil]\!],\mathfrak{M}[\![Node]\!]]\circ in_2\circ(f\times f) && \text{(Lemma 4.13)}\\ &= \mathfrak{M}[\![Node]\!]\circ(f\times f) && \text{(Lemma 4.13).} \end{aligned}$$

Allgemein ist mit genau der gleichen Argumentation eine Σ-Algebra eine Abbildung

$$\alpha : \sum_{f/n\in\Sigma} M.\to M$$

(wobei wir die übliche Summennotation auf unsere Mengenkonstruktion + ausdehnen, ebenso gleich für Abbildungen), und gegeben zwei solche Σ-Algebren α,β mit Trägermengen M bzw. N ist eine Abbildung $h : M\to N$ ein Homomorphismus genau dann, wenn

$$\begin{array}{ccc} \sum_{f/n\in\Sigma} M^n & \xrightarrow{\sum_{f/n\in\Sigma} h^n} & \sum_{f/n\in\Sigma} N^n \\ {\scriptstyle \alpha}\downarrow & & \downarrow{\scriptstyle \beta} \\ M & \xrightarrow[f]{} & N \end{array}$$

kommutiert. Wir schreiben kurz F_Σ für die durch Σ wie oben induzierte Konstruktion auf Mengen und Abbildungen, also

$$F_\Sigma M = \sum_{f/n\in\Sigma} M^n \quad F_\Sigma h = \sum_{f/n\in\Sigma} h^n;$$

dann wird das Diagramm zu

$$\begin{array}{ccc} F_\Sigma M & \xrightarrow{F_\Sigma h} & F_\Sigma N \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ M & \xrightarrow[f]{} & N \end{array}$$

4.1 Initialität und Rekursion

Wie der Beweis der Initialität der Termalgebra schon andeutet, ist Initialität einfach die abstrakte Verkapselung eines Rekursionsprinzips: Gegeben eine Signatur Σ mit Semantik (also Termalgebra) $[\![\Sigma]\!]$ können wir eine Funktion $h : [\![\Sigma]\!] \to A$ in eine Menge A definieren per

$$h(f(x_1, \ldots, x_n)) = a_f(h(x_1), \ldots, h(x_n)), \tag{9}$$

wobei wir für jedes $f/n \in \Sigma$ eine Funktion $a_f : A^n \to A$ wählen. Die Wahl der a_f definiert nämlich eine Σ-Algebra mit Träger A, und Gleichung (9) besagt einfach, dass h ein Homomorphismus nach A ist.

Dieses Rekursionsprinzip bezeichnet man in der funktionalen Programmierung als *fold*. Im Falle von *Tree* z.B. muss man auf A eine Konstante c (für *Nil*) und eine zweistellige Funktion g (für *Node*) angeben; wenn wir die dadurch gegebene Funktion $[\![\mathit{Tree}]\!] \to A$ mit *fold c g* bezeichnen (bzw. eben gleich aus *fold* eine Funktion höherer Stufe mit Argumenten c, g machen), lautet die rekursive Definition von *fold c g*

$$\begin{aligned} \mathit{fold}\ c\ g\ \mathit{Nil} &= c \\ \mathit{fold}\ c\ g\ (\mathit{Node}\ t\ s) &= g\ (\mathit{fold}\ c\ g\ t)\ (\mathit{fold}\ c\ g\ s) \end{aligned}$$

Z.B. liefert

$$\mathit{fold}\ 1\ +$$

eine Funktion, die die Anzahl Blätter eines Baums zählt.

Das fold-Schema ist zunächst einmal schwächer als die uns bereits geläufige *primitive Rekursion*, wie man sie etwa in der Definition der Fakultätsfunktion verwendet:

$$\begin{aligned} \mathit{fact}\ 0 &= 1 \\ \mathit{fact}\ (\mathit{Suc}\ n) &= (\mathit{Suc}\ n) \cdot (\mathit{fact}\ n). \end{aligned}$$

Man beachte, dass in der zweiten Zeile rechts nicht nur, wie im fold-Schema, auf das Ergebnis des rekursiven Aufrufs *factn* zugegriffen wird, sondern auch auf das Konstruktorargument n selbst. Trotzdem können wir die Fakultätsfunktion im fold-Schema programmieren. Dazu ändern wir den Ergebnistyp von $\mathbb{N}$ auf $\mathbb{N} \times \mathbb{N}$, wobei die zweite Komponente einfach das Argument zurückgibt; d.h. wir programmieren statt *fact* die Funktion $\langle \mathit{fact}, \mathit{id} \rangle$:

$$\begin{aligned} \langle \mathit{fact}, \mathit{id} \rangle\ 0 &= (1, 0) \\ \langle \mathit{fact}, \mathit{id} \rangle\ (\mathit{Suc}\ n) &= (\mathit{Suc}\ (\mathit{snd}\ (\langle \mathit{fact}, \mathit{id} \rangle\ n)) \cdot \mathit{fst}\ (\langle \mathit{fact}, \mathit{id}\ \rangle\ n), \mathit{Suc}\ (\mathit{snd}\ (\langle \mathit{fact}, \mathit{id} \rangle\ n))) \end{aligned}$$

(wobei wir im Einklang mit üblicher Progammierpraxis $fst = \pi_1$, $snd = \pi_2$ schreiben). Dieser Trick klappt natürlich auch im allgemeinen Fall, d.h. wir können das fold-Schema um die Möglichkeit erweitern, auf der rechten Seite von Definitionen die Konstruktorargumente zu verwenden.

Des weiteren möchten wir oft Funktionen mit mehreren Argumenten durch Rekursion über nur eines dieser Argumente (sagen wir, das erste) definieren, wie etwa die Konkatenation von Listen:

$$\begin{aligned} concat\ Nil\ k &= k \\ concat\ (Cons\ x\ l)\ k &= Cons\ x\ (concat\ l\ k). \end{aligned}$$

Mittels höherer Typen und Currying können wir dies als Spezialfall des bisherigen Formats ansehen: Wir definieren eben nicht $++ : List\ a \times List\ a \to List\ a$, sondern

$$++ : List\ a \to (List\ a \to List\ a).$$

Wir können dann beispielsweise die obige Definition umschreiben zu

$$\begin{aligned} concat\ Nil &= \lambda k.\, k \\ concat\ (Cons\ x\ l) &= \lambda k.\, Cons\ x\ (concat\ l\ k). \end{aligned}$$

Zusammenfassend haben wir

Definition 4.15. Eine *primitiv rekursive Definition* einer Funktion h besteht aus je einer Gleichung der Form

$$h(f(x_1, \ldots, x_n), y_2, \ldots, y_n) = g_f(h(x_1), \ldots, h(x_n), x_1, \ldots, x_n, y_2, \ldots, y_n)$$

für jeden Konstruktor $f/n \in \Sigma$. Dabei nennen wir den Ausdruck $f(x_1, \ldots, x_n)$ ein *Konstruktor-Pattern*.

Satz 4.16. *Eine primitiv rekursive Definition von h definiert h eindeutig.*

(Genauer besagt der Satz sogar, dass h darstellbar ist in einer Termsprache, die λ-Abstraktion und Applikation, Paare und Projektionen enthält.)

4.2 Mehrsortigkeit

Oft will man mehrere Datentypen durch *gegenseitige Rekursion* definieren, wie etwa den folgenden Typ von Bäumen unbegrenzten Verzweigungsgrads (*Rose Trees*):

Beispiel 4.17.

```
1 data Tree a, Forest a
2         Leaf: a -> Tree a
3         Node: Forest a -> Tree a
4         Nil: () -> Forest a
5         Cons: Tree a * Forest a -> Forest a
6         -- oder Cons: Tree a -> Forest a -> Forest a
```

Eine solche Deklaration definiert eine *sortierte Signatur*, hier mit Sortenmenge

$$S = \{a, \mathit{Tree}\ a, \mathit{Forest}\ a\}$$

(wobei wir a als *Parametersorte* markieren).

Die allgemeine Form ist wie folgt:

Definition 4.18. Eine *sortierte Signatur* $\Sigma = (S_0, S, F)$ besteht aus:

- Einer Menge S von *Sorten*
- Einer Menge $S_0 \subseteq S$ von *Parametern*
- Einer Menge F von Funktionssymbolen bzw. *Konstruktoren* c mit *Profilen* $c : a_1 \times \cdots \times a_n \to b$ mit $n \geq 0$, $a_1, \ldots, a_n \in S$ und $b \in S \setminus S_0$

Wir weisen nun, ähnlich wie im getypten λ-Kalkül, Termen Sorten zu:

Definition 4.19.

- Ein *Kontext* ist eine Menge $\Gamma = \{x_1 : a_1,\ \ldots,\ x_k : a_k\}$ mit Sorten $a_i \in S$ und paarweise verschiedenen Variablen x_i.
- *Sortierte Terme* im Kontext Γ werden induktiv definiert durch die Regeln

$$(x : a \in \Gamma)\ \ \frac{}{\Gamma \vdash x : a}$$

$$\frac{\Gamma \vdash t_1 : a_1 \qquad \ldots \qquad \Gamma \vdash t_n : a_n}{\Gamma \vdash c(t_1, \ldots, t_n) : b}$$

  Wir schreiben

$$T_\Sigma(\Gamma)_a = \{t \text{ Term } \mid \Gamma \vdash t : a\}$$

  für die Menge der Terme der Sorte a im Kontext Γ.

Entsprechend haben wir auch mehrsortige semantische Begriffe:

Definition 4.20. Sei $V = (V_a)_{a \in S_0}$ eine Mengenfamilie. Eine (mehrsortige) Σ-*Algebra über* V besteht aus

- einer Menge $\mathfrak{M}[\![a]\!]$ für jedes $a \in S$, mit $\mathfrak{M}[\![a]\!] = V_a$ für $a \in S_0$;
- einer Abbildung

$$\mathfrak{M}[\![c]\!] : \mathfrak{M}[\![a_1]\!] \times \cdots \times \mathfrak{M}[\![a_n]\!] \to \mathfrak{M}[\![b]\!]$$

  für jedes $c : a_1 \times \cdots \times a_n \to b$ in Σ.

Ein Σ-*Homomorphismus* $g : \mathfrak{M} \to \mathfrak{N}$ zwischen Σ-Algebren $\mathfrak{M}, \mathfrak{N}$ über V ist eine Familie $g = (g_a)_{a \in S}$ von Abbildungen

$$g_a : \mathfrak{M}[\![a]\!] \to \mathfrak{N}[\![a]\!] \qquad \text{mit } g_a = id \text{ für } a \in S_0.$$

Wir definieren einen Kontext $\Gamma(V) = \{x : a \mid a \in S_0, x \in V_a\}$; die *Termalgebra* $[\![\Sigma]\!]_V$ *über* V ist dann gegeben durch

$$\begin{aligned} [\![\Sigma]\!]_V[\![a]\!] &= T_\Sigma(\Gamma(V))_a && (a \in S) \\ [\![\Sigma]\!]_V[\![c]\!](t_1, \ldots, t_n) &= c(t_1, \ldots, t_n). \end{aligned}$$

Man beachte, dass dies wirklich eine Σ-Algebra über V ist: für $a \in S_0$ haben wir $[\![\Sigma]\!]_V[\![a]\!] = T_\Sigma(\Gamma(V))_a = V_a$, da es keine Konstruktoren des Profils $\cdots \to a$ gibt und somit $T_\Sigma(\Gamma(V))_a$ nur aus den Variablen der Sorte a in $\Gamma(V)$ besteht.

Satz 4.21 (Initialität/Folding)**.** *Für alle* Σ*-Algebren* $\mathfrak{N}$ *über* V *existiert genau ein* Σ*-Homomorphismus* $g : [\![\Sigma]\!]_V \to \mathfrak{N}$ *über* V*.*

Beispiel 4.22. Sei Σ_{Tree} die im Eingangsbeispiel deklarierte mehrsortige Signatur. Wir definieren eine Σ_{Tree}-Algebra über $V_a = \mathbb{N}$ per

$$\begin{aligned} \mathfrak{N}[\![\mathit{Tree}\ a]\!] &= \mathbb{N} = \mathfrak{N}[\![\mathit{Forest}\ a]\!] \\ \mathfrak{N}[\![\mathit{Leaf}]\!](n) &= n \\ \mathfrak{N}[\![\mathit{Node}]\!](n) &= n \\ \mathfrak{N}[\![\mathit{Nil}]\!] &= 0 \\ \mathfrak{N}[\![\mathit{Cons}]\!](n, m) &= n + m \end{aligned}$$

Per Initialität liefert dies einen eindeutigen Homomorphismus $h : [\![\Sigma]\!]_V \to \mathfrak{N}$ über V. Was berechnet dieser?

4.3 Strukturelle Induktion auf Datentypen

Als Beweisprinzip für rekursive Funktionen bietet sich typischerweise Induktion an; dabei sollte das verwendete Induktionsprinzip dieselbe Struktur haben wie die rekursive Definition. Wir erinnern uns z.B. an die Definition von Konkatenation:

Beispiel 4.23.

```
1 data List a where
2         Nil: () -> List a
3         Cons: a -> List a -> List a
4
5 concat: List a -> List a -> List a
6 concat Nil k = k
7 concat (Cons x l) k = Cons x (concat l k)
```

Hier wird primitive Rekursion über das erste Argument betrieben, d.h. die Definition von $concat\ (Cons\ x\ l)\ k$ wird zurückgeführt auf die von $concat\ l\ k$; das zugehörige Induktionsprinzip ist strukturelle Induktion über das erste Argument, d.h. Zurückführung der Induktionsbehauptung für $Cons\ x\ l$ (und k) auf die für l (und k). Wenn wir also z.B. die Assoziativität von *concat*

$$concat\ l\ (concat\ k\ v) = concat\ (concat\ l\ k)\ v$$

beweisen wollen, induzieren wir über l, wie folgt:

Nil: Wir haben $concat\ Nil\ (concat\ k\ v) = concat\ k\ v = concat\ (concat\ Nil\ k)\ v = concat\ k\ v$.

Cons x l: Wir haben

$$\begin{aligned} concat\ (Cons\ x\ l)\ (concat\ k\ v) &= Cons\ x(concat\ l(concat\ k\ v)) \\ &= Cons\ x\ (concat\ (concat\ l\ k)\ v) \quad \text{(IV)} \quad =: (\#) \end{aligned}$$

sowie

$$\begin{aligned} concat\ (concat\ (Cons\ x\ l)\ k)\ v &= concat(Cons\ x\ (concat\ l\ k))\ v \\ &= Cons\ x\ (concat\ (concat\ l\ k)\ v) \qquad = (\#). \end{aligned}$$

Nicht sehr erfolgversprechend ist dagegen zum Beispiel die Induktion über k: Wir können im Fall für $Cons\ x\ k$ zwar die linke Seite der Induktionsbehauptung umformen per

$$concat\ l\ (concat\ (Cons\ x\ k)\ v) = concat\ l\ (Cons\ x\ (concat\ k\ v)),$$

aber da bleiben wir stecken; die rechte Seite $concat\ (concat\ l\ (Cons\ x\ k))\ v$ lässt sich zunächst überhaupt nicht weiter umformen.

4.4 Induktion über mehrsortige Datentypen

Primitive Rekursion und strukturelle Induktion hat man natürlich auch im Mehrsortigen; allerdings gibt es dabei eine Besonderheit zu beachten. Wir erinnern an unsere Deklaration unbeschränkt verzweigender Bäume:

```
1 data Tree a, Forest a where
2         Leaf: a -> Tree a
3         Node: Forest a -> Tree a
4         Nil: () -> Forest a
5         Cons: Tree a -> Forest a -> Forest a
```

Man definiert ein primitiv rekursive Funktion auf *Tree a* immer *gleichzeitig* mit einer auf *Forest a*; das nennt man *gegenseitige Rekursion*. Der Grund hierfür ist, dass man, wie im einsortigen Fall demonstriert, primitive Rekursion auf das fold-Schema, d.h. auf eindeutige Homomorphismen von der initialen Algebra in eine gegebene Algebra zurückführt; letztere sind ja *Familien* von Abbildungen, eine für jede Sorte. Als Beispiel definieren wir eine Funktion, die einen Baum spiegelt:

```
1 mirrort: Tree a -> Tree a
2 mirrorf: Forest a -> Forest a
3         mirrort (Leaf x) = Leaf x
4         mirrort (Node f) = Node (mirrorf f)
5         mirrorf Nil = Nil
6         mirrorf (Cons t f) = concat (mirrorf f) ( Cons (mirrort t) Nil )
```

Dabei ist *concat* auf *Forest a* analog definiert wie oben auf *List a*; formal muss man gleichzeitig eine Funktion auf *Tree a* definieren, die man beliebig wählen kann.

Wir definieren außerdem eine Funktion, die die Liste der Blätter eines Baums (von links nach rechts) berechnet:

```
1 flattent: Tree a -> List a
2 flattenf: Forest a -> List a
3         flattent (Leaf x) = [x]
4         flattent (Node f) = flattenf f
5         flattenf Nil = Nil
6         flattenf (Cons t f) = concat (flattenf t) (flattenf f)
```

Die letzteren Funktionen lassen sich im übrigen auch als *fold* schreiben:

$$(\mathit{flattent}, \mathit{flattenf}) = \mathit{fold}\ [\_]\ \mathit{id}\ \mathit{Nil}\ \mathit{concat}.$$

Wir wollen nun

$$\mathit{flattent}\ (\mathit{mirrort}\ t) = \mathit{rev}\ (\mathit{flattent}\ t)$$

für alle $t : \mathit{Tree}\ a$ zeigen, wobei *rev* auf Listen rekursiv definiert ist durch

$$\begin{aligned} \mathit{rev}\ \mathit{Nil} &= \mathit{Nil} \\ \mathit{rev}\ (\mathit{Cons}\ x\ l) &= \mathit{concat}\ (\mathit{rev}\ l)\ [x]. \end{aligned}$$

Hierzu verwenden wir strukturelle Induktion. Das bedeutet unter anderem, dass wir die Behauptung für *Node f* auf f zurückführen; f ist aber kein Baum, sondern ein Wald. Wir brauchen also eine *zweite* Induktionsbehauptung, eben für Wälder, um an dieser Stelle eine verwertbare Induktionsvoraussetzung zu haben. Wir behaupten also zusätzlich

$$\mathit{flattenf}\ (\mathit{mirrorf}\ f) = \mathit{rev}\ (\mathit{flattenf}\ f)$$

für alle $f : \mathit{Forest}\ a$. Wir haben dann vier Fälle in der Induktion:

Leaf x: Einerseits haben wir $\mathit{flattent}\ (\mathit{mirrort}\ (\mathit{Leaf}\ x)) = \mathit{flattent}\ (\mathit{Leaf}\ x) = [x]$. Andererseits gilt $\mathit{rev}\ (\mathit{flattent}\ (\mathit{Leaf}\ x)) = \mathit{rev}\ [x]$, und man errechnet leicht $\mathit{rev}\ [x] = [x]$.

Nil. Es gilt $\mathit{flattenf}\ (\mathit{mirrorf}\ \mathit{Nil}) = \mathit{flattenf}\ \mathit{Nil} = \mathit{Nil}$ und $\mathit{rev}\ (\mathit{flattenf}\ \mathit{Nil}) = \mathit{rev}\ \mathit{Nil} = \mathit{Nil}$.

Node f: Wir haben einerseits

$$\begin{aligned} \mathit{flattent}\ (\mathit{mirrort}\ (\mathit{Node}\ f)) &= \mathit{flattent}\ (\mathit{Node}\ (\mathit{mirrorf}\ f)) \\ &= \mathit{flattenf}\ (\mathit{mirrorf}\ f) \\ &= \mathit{rev}\ (\mathit{flattenf}\ f) \qquad \text{(IV für Forest)}, \end{aligned}$$

und andererseits $rev\ (\mathit{flattent}\ (\mathit{Node}\ f)) = rev\ (\mathit{flattenf}\ f)$

Cons t f: Wir haben (unter Einführung von Hilfsaussagen on-the-fly, die wir anschließend beweisen, sowie unter Verwendung von Listennotation für Wälder)

$$\begin{aligned}
&\mathit{flattenf}\ (\mathit{mirrorf}\ (\mathit{Cons}\ t\ f)) \\
&= \mathit{flattenf}\ (\mathit{concat}\ (\mathit{mirrorf}\ f)\ [\mathit{mirrort}\ t]) \\
&= \mathit{concat}\ (\mathit{flattenf}\ (\mathit{mirrorf}\ f))\ (\mathit{flattenf}\ [\mathit{mirrort}\ t]) && \text{(Lemma A)} \\
&= \mathit{concat}\ (\mathit{flattenf}\ (\mathit{mirrorf}\ f))\ (\mathit{flattent}\ (\mathit{mirrort}\ t)) && \text{(Lemma B)} \\
&= \mathit{concat}\ (\mathit{rev}\ (\mathit{flattenf}\ f))\ (\mathit{rev}\ (\mathit{flattent}\ t)) && \text{(IV für Tree/Forest)}
\end{aligned}$$

Andererseits haben wir auch

$$\begin{aligned}
\mathit{rev}\ (\mathit{flattenf}\ (\mathit{Cons}\ t\ f)) &= \mathit{rev}\ (\mathit{concat}\ (\mathit{flattent}\ t)\ (\mathit{flattenf}\ f)) \\
&= \mathit{concat}\ (\mathit{rev}\ (\mathit{flattenf}\ f))\ (\mathit{rev}\ (\mathit{flattent}\ t)) \qquad \text{(Übung)}
\end{aligned}$$

Es verbleiben unsere beiden Hilfsaussagen:

Lemma 4.24 (Lemma A)**.** *Es gilt* $\mathit{flattenf}\ (\mathit{concat}\ f\ g) = \mathit{concat}\ (\mathit{flattenf}\ f)\ (\mathit{flattenf}\ g)$

Beweis. Induktion über f:
Nil : Wir haben $\mathit{flattenf}\ (\mathit{concat}\ \mathit{Nil}\ g) = \mathit{flattenf}\ g$ und $\mathit{concat}\ (\mathit{flattenf}\ \mathit{Nil})\ (\mathit{flattenf}\ g) = \mathit{concat}\ \mathit{Nil}\ (\mathit{flattenf}\ g) = \mathit{flattenf}\ g$.
Cons t f: Wir haben

$$\begin{aligned}
&\mathit{flattenf}\ (\mathit{concat}\ (\mathit{Cons}\ t\ f)\ g) \\
&= \mathit{flattenf}\ (\mathit{Cons}\ t\ (\mathit{concat}\ f\ g)) \\
&= \mathit{concat}\ (\mathit{flattent}\ t)\ (\mathit{flattenf}\ (\mathit{concat}\ f\ g)) \\
&= \mathit{concat}\ (\mathit{flattent}\ t)\ (\mathit{concat}\ (\mathit{flattenf}\ f)\ (\mathit{flattenf}\ g)) && \text{(IV)}
\end{aligned}$$

sowie

$$\begin{aligned}
&\mathit{concat}\ (\mathit{flattenf}\ (\mathit{Cons}\ t\ f))\ (\mathit{flattenf}\ g) \\
&= \mathit{concat}\ (\mathit{concat}\ (\mathit{flattent}\ t)\ (\mathit{flattenf}\ f))\ (\mathit{flattenf}\ g),
\end{aligned}$$

was per Assoziativität von *concat* gleich der linken Seite ist. (Die Induktionsbehauptung für Bäume kann hier als $\top$ gewählt werden.) □

Lemma 4.25 (Lemma B)**.** *Es gilt* $\mathit{flattenf}\ [t] = \mathit{flattent}\ t$.

Beweis. Wir haben $\mathit{flattenf}\ (\mathit{Cons}\ t\ \mathit{Nil}) = \mathit{concat}\ (\mathit{flattent}\ t)\ (\mathit{flattenf}\ \mathit{Nil}) = \mathit{concat}\ (\mathit{flattent}\ t)\ \mathit{Nil} = \mathit{flattent}\ t$, wobei wir im letzten Schritt verwenden, dass *Nil* auch rechtsneutral bezüglich *concat* ist; dies zeigt man separat durch strukturelle Induktion. □

4.5 Kodatentypen

Wie wir oben gesehen haben, sind *Daten* (also Elemente eine Datentyps) gegeben über ihre *Konstruktion* – z.B. ist der Datentyp der Listen dadurch gegeben, dass *Nil* eine Liste ist, und wenn l eine Liste ist, dann auch *Cons* x l. Da man verlangt, dass die Konstruktion eines Datentypelements ein endlicher Prozess ist (genauer *wohlfundiert*), sind Elemente von Datentypen endliche Objekte, also z.B. endliche Listen.

Hierzu dual befassen wir uns nun mit *Kodaten*, die dadurch gegeben sind, wie sie *destruiert* oder *beobachtet* werden. Dies liefert dann potentiell unendliche Objekte, die wir uns typischerweise als *Prozesse* vorstellen.

Definition 4.26 (Streams)**.** Ein *Stream* über einem Alphabet A ist eine unendliche Sequenz

$$(a_0, a_1, a_2, \dots)$$

mit $a_i \in A$ für alle $i \in \mathbb{N}$. Die Menge der Streams über A bezeichnen wir mit A^ω. Es gibt zunächst keinen offensichtlichen Weg, einen Stream zu *konstruieren*: zwar kann ich an einen Stream ein Element vorn anhängen, aber das liefert offenbar kein terminierendes Konstruktionsverfahren für Streams. Dagegen kann man Streams in der von Listen gewohnten Weise *destruieren*: Man hat

$$\begin{aligned} hd : S &\to A \\ (a_0, a_1, \dots) &\mapsto a_0 \end{aligned}$$

und

$$\begin{aligned} tl : S &\to S \\ (a_0, a_1, \dots) &\mapsto (a_1, a_2, \dots). \end{aligned}$$

Dies macht gleichzeitig die Analogie zwischen Destruktoren und Beobachtungen klar: hd liest das aktuelle Element eines Streams, tl geht zum nächsten Element über.

Notation 4.27. Wir verwenden eine zu unserer Syntax für Datentypen duale Syntax für Codatentypen, in der wir einfach die Destruktoren eines Kodatentyps auflisten. Streams deklariert man dann in der Form

```
1 codata Stream where
2   hd: Stream -> A
3   tl: Stream -> Stream
```

Wir wollen nun dual zu primitiver Rekursion Funktionen nur mittels Zugriff auf den Kodatentyp über seine Destruktoren definieren. Während rekursive Funktionen Daten als *Eingabe* haben, haben solche *korekursiven* Funktionen Kodaten (dualerweise) als *Ausgabe*. Z.B. würden wir gerne eine Funktion $map\ f : A^\omega \to A^\omega$, die jedes hereinkommende Element des Streams mit einer Funktion $f : A \to A$ verarbeitet und wieder ausgibt, definieren per

$$\begin{aligned} hd\ (map\ f\ s) &= f\ (hd\ s) \\ tl\ (map\ f\ s) &= (map\ f)\ (tl\ s). \end{aligned}$$

Anders als im induktiven Fall ist aber für das bloße Auge hier nicht unmittelbar klar, dass so etwas in der Tat eine Definition darstellt. Diese Frage klären wir im folgenden.

Zunächst halten wir fest, dass wir die beiden Destruktoren hd, tl zu einer Abbildung

$$\langle hd, tl\rangle : A^\omega \to A \times A^\omega$$

zusammenfassen können. Wir sehen wiederum die Dualität zu Datentypen: Während eine Σ-Algebra eine Abbildung von einer aus der Grundmenge M konstruierten Menge nach M ist (im Falle von Σ_{Tree}-Algebren z.B. eine Abbildung $1 + M \times M \to M$), haben wir hier eine Abbildung von der Form $\alpha : M \to A \times M$, also von der Grundmenge M in eine aus ihr konstruierte Menge. Es liegt nahe, so eine Struktur als eine *Koalgebra* zu bezeichnen. Die natürliche Dualisierung der diagrammatischen Sichtweise auf Homomorphismen, die wir oben entwickelt haben, wäre, als *Morphismen* zwischen solchen Strukuren $\beta : N \to A \times N$, $\alpha : M \to A \times M$ solche Abbildungen $h : N \to M$ anzusehen, für die das Diagramm

$$\begin{array}{ccc} N & \xrightarrow{\ h\ } & M \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\alpha} \\ A \times N & \xrightarrow[A\times h]{} & A \times M \end{array}$$

kommutiert, wobei wir (in $A \times h$) kurz A statt id_A schreiben. Dann bedeutet die Definition von *map* f gerade, dass *map* f ein Homomorphismus von der durch

$$\beta(s) = (f\ (hd\ s), tl\ s) : A^\omega \to A \times A^\omega$$

gegebenen Koalgebra in die beabsichtigte Interpretation unseres Stream-Kodatentyps, also die durch $\alpha = \langle hd, tl\rangle : A^\omega \to A \times A^\omega$ gegebene Koalgebra, ist. Diese Idee werden wir nun in etwas allgemeinerem Rahmen formalisieren, dabei allerdings den Einstieg in kategorientheoretische Begrifflichkeit fürs erste vermeiden.

Wir erinnern uns an die Konstruktionen $+$ und $\times$, die sowohl auf Mengen als auch auf Abbildungen funktionieren. Wir definieren nun mittels dieser Konstruktionen eine Klasse von Ausdrücken, die wir *Mengenoperatoren* nennen, mittels der Grammatik

$$G ::= A \mid id \mid G_1 + G_2 \mid G_1 \times G_2 \qquad (A \text{ Menge}).$$

Wir definieren die Anwendung GX von G auf eine Menge X rekursiv durch

$$\begin{aligned} AX &= A \\ idX &= X \\ (G_1 + G_2)X &= G_1X + G_2X \\ (G_1 \times G_2)X &= G_1X \times G_2X. \end{aligned}$$

Ganz analog definieren wir die Anwendung Gf von G auf eine Funktion $f : X \to Y$, wobei wir wieder A mit id_A verwechseln. Damit haben wir insbesondere

$$Gf : GX \to GY.$$

In Anwendungen definieren wir G meist durch die Angabe von GX.

Bemerkung 4.28. Zu einer (Datentyp-)Signatur Σ definieren wir den Mengenoperator

$$F_\Sigma X = \sum_{f/n \in \Sigma} X^n,$$

Wie wir oben gesehen haben, entsprechen Σ-Algebren dann Abbildungen der Form $F_\Sigma M \quad \to M$. Im Falle der binären Bäume hatten wir z.B.

$$F_{\Sigma_{Tree}} M = 1 + M^2.$$

Dual dazu haben wir

Definition 4.29. Eine *G-Koalgebra* für einen Mengenoperator G ist eine Abbildung $M \to GM$. Die Elemente von M nennen wir *Zustände*.

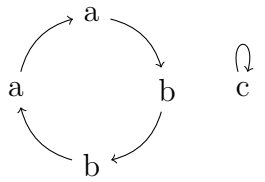
Beispiel 4.30 (Streams)**.** Wenn wir annehmen, dass die Parametersorte a durch die Menge A interpretiert wird, gehört zur Kodatentypdeklaration *Stream a* der Mengenoperator

$$GX = A \times X.$$

Eine G-Koalgebra $\mathfrak{M}$ ist dann eine Abbildung der Form $\alpha : M \to A \times M$, z.B.

$$\langle hd, tl \rangle : A^\omega \to A \times A^\omega.$$

Für jeden Zustand $x \in M$ haben wir also ein Paar $\alpha(x) = (\mathfrak{M}[\![hd]\!](x), \mathfrak{M}[\![tl]\!](x))$, bestehend aus einem Output $\mathfrak{M}[\![hd]\!](x) \in A$ und einem Nachfolgezustand $\mathfrak{M}[\![tl]\!](x))$. Wenn wir die Outputs direkt an die Zustände schreiben und die Nachfolgerabbildung durch Pfeile darstellen, stellt also z.B.



eine G-Koalgebra dar.

Wir erinnern daran, dass der durch Σ definierte Datentyp gerade die initiale Σ-Algebra ist. Dual dazu wollen wir den durch G definierten Kodatentyp als die *finale* G-Koalgebra begreifen. Dazu benötigen wir, wie schon angekündigt, einen geeigneten Morphismenbegriff.

Mit der gerade eingeführten Schreibweise für Mengenoperatoren ist eine Abbildung $f : \mathfrak{M} \to \mathfrak{N}$ ein Σ-Homomorphismus zwischen Σ-Algebren $\mathfrak{M}$, $\mathfrak{N}$ genau dann, wenn folgendes Diagramm kommutiert:

$$\begin{array}{ccc} F_\Sigma M & \xrightarrow{F_\Sigma f} & F_\Sigma N \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ M & \xrightarrow[f]{} & N \end{array}$$

Dual dazu legen wir fest, dass $f : \mathfrak{M} \to \mathfrak{N}$ ein *G-Koalgebramorphismus* (oder kurz *Morphismus*) zwischen G-Koalgebren $\mathfrak{M}$, $\mathfrak{N}$ ist, wenn das Diagramm

$$\begin{array}{ccc} M & \xrightarrow{f} & N \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ GM & \xrightarrow[Gf]{} & Gn \end{array}$$

kommutiert.

Beispiel 4.31 (Streams)**.** Für $GX = A \times X$ ist $f : \mathfrak{M} \to \mathfrak{N}$ ein G-Koalgebramorphismus zwischen G-Koalgebren $\mathfrak{M}$, $\mathfrak{N}$ genau dann, wenn

$$\begin{array}{ccc} M & \xrightarrow{f} & N \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ A \times M & \xrightarrow[A\times f]{} & A \times N \end{array}$$

kommutiert. Wenn wir $\alpha = \langle \mathfrak{M}[\![hd]\!], \mathfrak{M}[\![tl]\!] \rangle$ und $\beta = \langle \mathfrak{N}[\![hd]\!], \mathfrak{N}[\![tl]\!] \rangle$ schreiben, dann bedeutet dies, dass

$$\begin{aligned} \mathfrak{N}[\![hd]\!](f(x)) &= \mathfrak{M}[\![hd]\!](x) \qquad \text{und} \\ \mathfrak{N}[\![tl]\!](f(x)) &= f(\mathfrak{M}[\![tl]\!](x)). \end{aligned}$$

Dies bedeutet, dass f das *Verhalten* von Zuständen bewahrt: $f(x)$ hat denselben Output wie x, und f kommutiert mit der Nachfolgerfunktion tl.

Dual zum Begriff der initialen Algebra haben wir:

Definition 4.32. Eine G-Koalgebra $\mathfrak{N}$ ist *final*, wenn für jede G-Koalgebra $\mathfrak{M}$ genau ein G-Koalgebramorphismus $\mathfrak{M} \to \mathfrak{N}$ existiert.

Satz 4.33. *Finale G-Koalgebren sind eindeutig bis auf Isomorphismus.*

Beweis. Dual zum Satz über die Eindeutigkeit der initialen Algebra. □

Es bleibt die Frage nach der Existenz finaler Koalgebren. Wir handeln zunächst den Spezialfall für Streams ab:

Satz 4.34. *Sei $GX = A \times X$. Die G-Koalgebra $\langle hd, tl \rangle : A^\omega \to A \times A^\omega$ ist final.*

Beweis. Sei $\mathfrak{N}$ G-Koalgebra; wir schreiben kurz $\mathfrak{N}[\![hd]\!] = hd_{\mathfrak{N}}$, $\mathfrak{N}[\![tl]\!] = tl_{\mathfrak{N}}$.

Für einen Zustand $x \in N$ setze $f(x) = (a_0, a_1, \dots)$ mit

$$a_i = hd_{\mathfrak{N}}(tl^i_{\mathfrak{N}}(x)).$$

Zu zeigen sind:

1. *f ist G-Koalgebramorphismus:* Wir haben

$$hd(f(x)) = a_0 = hd_{\mathfrak{N}}(x)$$

und

$$tl(f(x)) = (a_1, a_2, \dots) =: (b_0, b_1, \dots) \quad \text{mit } b_i = a_{i+1} = hd_{\mathfrak{N}}(tl_{\mathfrak{M}}^{i+1}(x)) = hd_{\mathfrak{M}}(tl_{\mathfrak{M}}^{i}(tl_{\mathfrak{M}}(x))),$$

also $tl(f(x)) = f(tl_{\mathfrak{N}}(x))$.

2. *f ist eindeutig:* Sei $g : \mathfrak{M} \to S$ ein G-Koalgebramorphismus. Wir zeigen per Induktion über i, dass $g(x)_i = f(x)_i$ für alle Zustände $x \in M$, wobei der Index i die i-te Position im Stream bezeichnet.

   $i = 0$: Wir haben

$$\begin{aligned} g(x)_0 &= hd(g(x)) \\ &hd_{\mathfrak{N}}(x) && \text{(g ist Koalgebramorphismus)} \\ &= f(x). \end{aligned}$$

   $i \to i + 1$: Wir haben

$$\begin{aligned} g(x)_{i+1} &= tl(g(x))_i \\ &= g(tl_{\mathfrak{N}}(x))_i && (g \text{ ist Koalgebramorphismus}) \\ &= f(tl_{\mathfrak{N}}(x))_i && \text{(IV)} \\ &= tl(f(x))_i && (f \text{ ist Koalgebramorphismus}) \\ &= f(x)_{i+1} \end{aligned}$$

□

Definition 4.35 (Unfold)**.** Für $h : M \to A$, $t : M \to M$ wird somit eine Funktion unfold $ht : M \to A^\omega$ definiert per

$$\begin{aligned} hd(\text{unfold } h\ t\ x\ ) &= h\ x \\ tl(\text{unfold } h\ t\ x\ ) &= \text{unfold } h\ t\ (t\ x) \end{aligned}$$

Beispiel 4.36. Wir können einen Stream *ones*, also eine Abbildung $ones : 1 \to A^\omega$, corekursiv definieren durch

$$\begin{aligned} hd\ ones &= 1 \\ tl\ ones &= ones, \end{aligned}$$

oder in einer Zeile per $ones = \text{unfold } (\lambda x.1) id_1$.

Beispiel 4.37. Wir wollen eine Funktion $\mathit{blink} : a \times \mathit{Stream}\ a \to \mathit{Stream}\ a$ definieren, so dass $\mathit{blink}\ x\ (a_0, a_1, \dots) = (x, a_0, x, a_1, x, a_2, \dots)$:

$$\begin{aligned} \mathit{hd}\ (\mathit{blink}\ x\ s) &= x \\ \mathit{tl}\ (\mathit{blink}\ x\ s) &= ? \end{aligned}$$

Die rechte Seite lässt sich durch *blink* nicht geeignet ausdrücken, da sie von der Form $(a_0, x, a_1, \dots)$ sein müsste. wir brauchen also eine zweite Abbildung, die x an den ungeraden Positionen in den Stream einfügt. Dazu erweitern wir den Definitionsbereich von *blink* zu

$$\mathit{blink} : 2 \times a \times \mathit{Stream}\ a \to \mathit{Stream}\ a,$$

wobei 2 der Datentyp mit Konstruktoren 0, 1 ist; wir wollen das Verhalten

$$\mathit{blink}\ c\ x\ (a_0, a_1, \dots) = \begin{cases} (x, a_0, x, a_1, \dots) & \text{falls } c = 0 \\ (a_0, x, a_1, x, \dots) & \text{falls } c = 1 \end{cases}$$

implementieren. Wir schreiben kurz $\mathit{blink}\ 0 = b_0$, $\mathit{blink}\ 1 = b_1$ und definieren dann b_0, b_1 per unfold:

$$\begin{aligned} \mathit{hd}\ (b_0\ x\ s) &= x \\ \mathit{tl}\ (b_0\ x\ s) &= b_1\ x\ s \\ \mathit{hd}\ (b_1\ x\ s) &= \mathit{hd}\ s \\ \mathit{tl}\ (b_1\ x\ s) &= b_0\ x\ (\mathit{tl}\ s) \end{aligned}$$

Bemerkung 4.38. Wie das Stream-Beispiel illustriert, können wir die Elemente der finalen G-Koalgebra als die möglichen Verhalten von Zuständen in G-Koalgebren ansehen. In diesem Sinne bildet der eindeutige Morphismus von einer G-Koalgebra in die finale G-Koalgebra einen Zustand auf sein *Verhalten* ab. In diesen Begriffen haben wir:

1. Jeder Zustand in der finalen Koalgebra ist sein eigenes Verhalten (denn: die Identität ist der eindeutige Morphismus der finalen Koalgebra in sich selbst).
2. Morphismen von G-Koalgebren bewahren Verhalten (denn: wenn $f : N_1 \to N_2$ ein Morphismus von G-Koalgebren ist, M die finale G-Koalgebra, und $g : N_2 \to M$, dann ist $g \circ f$ der eindeutige Morphismus $N_1 \to M$).

4.6 Koinduktion

Nachdem wir als zur Rekursion duales Definitionsprinzip die Korekursion eingeführt haben, benötigen wir nunmehr ein geeignetes Beweisprinzip, um Aussagen über korekursive Funktionen herzuleiten. Als Beispiel betrachten wir folgende korekursive Definition zweier Funktionen, die aus einem Stream die Teilstreams an den geraden bzw. ungeraden Positionen herausgreifen:

$$\begin{aligned} hd\ (even\ s) &= hd\ s \\ hd\ (odd\ s) &= hd\ (tl\ s) \\ tl\ (even\ s) &= even\ (tl\ (tl\ s)) \\ tl\ (odd\ s) &= odd\ (tl\ (tl\ s)) \end{aligned}$$

Mit obiger Definition von *blink* würden wir dann z.B. gerne zeigen, dass

$$odd\ (blink\ 0\ x\ s) = s$$

gilt (da ja *blink* 0 das Füllsymbol x an den geraden Positionen einfügt). Wir verifizieren leicht, dass dies an Position 0 stimmt:

$$hd\ (odd\ (blink\ 0\ x\ s)) = hd\ (tl\ (blink\ 0\ x\ s)) = hd(\ (blink\ 1\ x\ s)) = hd\ s.$$

Außerdem können wir untersuchen, was mit dem Rest des Streams passiert:

$$\begin{aligned} tl\ (odd\ (blink\ 0\ x\ s)) &= odd\ (tl\ (tl\ (blink\ 0\ x\ s))) \\ &= odd\ (tl\ (blink\ 1\ x\ s)) \\ &= odd\ (blink\ 0\ x\ (tl\ s)). \end{aligned}$$

Wenn wir nun unterstellen, dass $blink\ 0\ x\ (tl\ s) = tl\ s$, dann hätten wir nunmehr gezeigt, dass auch die Reststreams gleich sind, d.h.

$$tl\ (odd\ (blink\ 0\ x\ s)) = tl\ s,$$

woraus ja wohl folgen würde, dass insgesamt $odd\ (blink\ 0\ x\ s) = s$; die Behauptung für s auf die für $tl\ s$ zu „reduzieren“, hört sich aber nicht unmittelbar nach einer korrekten Beweismethode an. Trotzdem ist diese Art *zirkulärer* Beweis korrekt, wenn man bestimmte Restriktionen einhält. Wir formalisieren dies für Streams wie folgt:

Definition 4.39. Eine Relation $R \subseteq A^\omega \times A^\omega$ heißt *Bisimulation*, wenn für alle $(s,t) \in R$ gilt:

- $hd\ s = hd\ t$
- $(tl\ s)\ R\ (tl\ t)$

Satz 4.40 (Koinduktion)**.** *Wenn R eine Bisimulation ist, dann gilt $R \subseteq id$, d.h.*

$$sRt \implies s = t$$

für all $s, t \in A^\omega$.

Beweis. Sei sRt; wir schreiben $s = (x_0, x_1, \dots)$, $t = (y_0, y_1, \dots)$. Wir zeigen

$$(x_i, x_{i+1}, \dots) R (y_i, y_{i+1}, \dots) \qquad ((*))$$

per Induktion über $i \geq 0$:

$i = 0$: OK nach Voraussetzung.

$i \to i + 1$: Nach Induktionsvoraussetzung gilt $(x_i, x_{i+1}, \dots) R (y_i, y_{i+1}, \dots)$. Da R Bisimulation ist, folgt $tl\ (x_i, x_{i+1}, \dots) R tl\ (y_i, y_{i+1}, \dots)$, also $(x_{i+1}, x_{i+2}, \dots) R (y_{i+1}, y_{i+2}, \dots$.

Aus $(*)$ folgt, da R eine Bisimulation ist,

$$x_i = hd\ (x_i, x_{i+1}, \dots) = hd (y_i, y_{i+1}, \dots) = y_i.$$

Somit gilt $s = t$. □

Bemerkung 4.41. Man kann obigen Satz als *Korrektheit* des Bisimulationsprinzips auffassen: Wenn ich zwei Streams durch eine Bisimulation in Beziehung setzen kann, sind sie tatsächlich gleich. Die Umkehrung ist klar. Es gilt trivialerweise auch *Vollständigkeit*, also die Umkehrung:: Wenn $s = t$, dann existiert eine Bisimulation R mit sRt, nämlich $R = id$. (Man überzeugt sich leicht, dass id in der Tat eine Bisimulation ist.)

Beispiel 4.42. Mit Hilfe des Bisimulationsprinzips führen wir jetzt den Beweis, dass in der Tat $odd\ (blink\ 0\ x\ s) = s$ gilt. Dazu müssen wir eine geignete Bisimulation erfinden. Wir versuchen es mit $R = \{(odd\ (f\ x\ s), s) \mid x \in A, s \in A^\omega\}$.

1. $hd\ (odd\ (f\ x\ s)) = hd\ (tl\ (f\ x\ s)) = hd\ (g\ x\ s) = hd\ s$ ✓
2. $tl\ (odd\ (f\ x\ s)) = odd\ (tl\ (tl\ (f\ x\ s))) = odd\ (tl\ (g\ x\ s)) = odd\ (f\ x\ (tl\ s)) R tl\ s$ ✓

also ist $odd\ (f\ x\ s) = s \forall x, s$.

Beispiel 4.43. Anschauung: $zip\ (x_0, \dots)(y_0, \dots) = (x_0, y_0, x_1, y_1, \dots)$

$hd\ (zip\ s\ t) = hd\ s$

$tl\ (zip\ s\ t) = zip\ t\ (tl\ s)$

Behauptung: $zip\ (even\ s)(odd\ s) = s$

Beweis. Ist $R = \{(zip\ (even\ s)(odd\ s), s) \mid s \in A^\omega\}$ eine Bisimulation?

1. $hd\ (zip\ (even\ s)(odd\ s)) = hd\ (even\ s) = hd\ s$ ✓
2. $tl\ (zip\ (even\ s)(odd\ s)) = zip\ (odd\ s)(tl\ even\ s) = zip\ (odd\ s)(even\ (tl\ (tl\ s)))$. Es ist unklar, ob das in R-Relation zu $tl\ s$ steht.

Die Relation R scheint zu klein gewählt. Setze also

$R' = R \cup \{(zip\ (odd\ s)(even\ (tl\ (tl\ s))), tl\ s) \mid s \in A^\omega\}$

$hd\ (zip\ (odd\ s)(\dots)) = hd\ (odd\ s) = hd\ (tl\ s))$ ✓

$tl\ (zip\ (odd\ s)(even\ (tl\ (tl\ s))) = zip\ (even\ (tl\ (tl\ s)))(tl\ (odd\ s))$

$= zip\ (even\ (tl\ (tl\ s)))(odd\ (tl\ (tl\ s)))\quad R'\quad tl\ (tl\ s)$ □

4.7 Kodatentypen mit Alternativen

Wir betrachten das folgende Beispiel eines Kodatentyps mit Alternativen:

```
1 codata IList a where
2         end: IList a @ dead -> ()
3         hd:  IList a @ alive -> a
4         tl:  IList a @ alive -> IList a
```

Diese Spezifikation definiert die finale G-Koalgebra für den Mengenoperator $GX = 1 + A \times X$ (mit $A = [\![a]\!]$), d.h. $\alpha : N \to 1 + A \times N$.
Hieraus ergibt sich die disjunkte Zerlegung

$$N = \underbrace{\alpha^{-1}[1]}_{\text{dead}} \cup \underbrace{\alpha^{-1}[A \times N]}_{\text{alive}},$$

so dass

$$\begin{aligned} \mathit{end} &: \mathit{dead} \to 1 \\ \langle \mathit{hd}, \mathit{tl} \rangle &: \mathit{alive} \to A \times N \end{aligned}$$

Wir definieren allgemein:

Definition 4.44. Für $GX = \sum_{i=1}^{n} A_i \times X^{k_i}$ beschreibe die finale G-Koalgebra durch die *Signatur* Σ aus

- Alternativen $d_1, \ldots, d_n$ (im Beispiel: $d_1 = \mathit{dead}, d_2 = \mathit{alive}$),
- für $i = 1, \ldots, n$ je $k_i + 1$ *Observer*

  $$t_{ij} : d_i \to d \text{ und } h_i : d_i \to a_i$$

  wobei $[\![a_i]\!] = A_i$ und $j = 1, \ldots, k_i$ (im Beispiel: *kein* t_{1j}, $h_1 = \mathit{end}$, $t_{21} = \mathit{tl}$, $h_2 = \mathit{hd}$).

Die hierdurch definierte Σ-Koalgebra $\mathfrak{N}$ besteht dann aus

- einer Trägermenge N,
- einer disjunkten Zerlegung $N = \bigcup_{i=1}^{n} \mathfrak{N}[\![d_i]\!]$,
- Interpretationen aller Observer: $\mathfrak{N}[\![t_{ij}]\!] : \mathfrak{N}[\![d_i]\!] \to N$ und $\mathfrak{N}[\![h_i]\!] : \mathfrak{N}[\![d_i]\!] \to A_i$.

Definition 4.45. Eine Funktion $f : M \to N$ ist ein Σ-Homomorphismus (schreibe: $f : \mathfrak{M} \to \mathfrak{N}$), wenn sie die folgenden Eigenschaften erfüllt:

- $f[\mathfrak{M}[\![d_i]\!]] \subseteq \mathfrak{N}[\![d_i]\!]$
- $\mathfrak{N}[\![h_i]\!](f(x)) = \mathfrak{M}[\![h_i]\!](x)$
- $\mathfrak{N}[\![t_{ij}]\!](f(x)) = f(\mathfrak{M}[\![t_{ij}]\!](x))$

Hieraus ergibt sich erneut das Konzept der *finalen* Σ-Koalgebra (nun mit Alternativen), d.h. einer Koalgebra, in die von jeder Koalgebra aus genau ein Homomorphismus existiert. Im Allgemeinen sind Elemente solcher finalen Koalgebren unendliche Bäume in denen jeder Knoten sowohl mit einer Alternative d_i als auch mit einem Element $x \in A_i$ beschriftet ist und über k_i Nachfolgeknoten verfügt (also über einen Nachfolgeknoten pro t_{ij}).
Für den oben betrachteten Kodatentypen `IList a` besteht die finale Koalgebra beispielsweise aus allen endlichen oder unendlichen Listen von Elementen aus a.

Beispiel 4.46. Nun ist es möglich, Funktionen mittels Korekursion zu definieren. Beispielsweise können wir eine Funktion $takeUntil : a \times IList\, a \to IList\, a$ definieren, so dass $takeUntil(x, s)$ eine Liste liefert, die sich so lange wie s verhält, bis das Element x erreicht wird. Falls das Element x erreicht wird, soll $takeUntil(x, s)$ sofort enden.

Hierzu definieren wir zunächst die folgende disjunkte Zerlegung von $takeUntil(x, s)$:

> Wenn $s@alive$, dann $takeUntil(x, s)@dead \Leftrightarrow hd\, s = x$ und
> wenn $s@dead$, dann $takeUntil(x, s)@dead$.

Sodann definieren wir für alle x und s mit $takeUntil(x, s)@alive$:

$$hd(takeUntil(x, s)) = hd\, s$$
$$tl(takeUntil(x, s)) = takeUntil(x, (tl\, s))$$

Beispiel 4.47. Wir betrachten nun die folgende Spezifikation eines weiteren Kodatentyps mit Alternativen:

```
1 codata Terrain a where
2         objects: Terrain a -> List a
3         deadend: Terrain a @ stuck -> ()
4         left:    Terrain a @ junction -> Terrain a
5         right:   Terrain a @ junction -> Terrain a
```

Der zu dieser Signatur gehörige Mengenoperator ist

$$GX = \underbrace{A^*}_{\text{objects}} \times (\underbrace{1}_{\text{deadend}} + \underbrace{X}_{\text{left}} \times \underbrace{X}_{\text{right}}) = A^* \times (1 + X^2),$$

wobei $A = [\![a]\!]$ und A^* die Menge aller endlichen Listen über A bezeichnet.

Die finale Koalgebra für G enthält also alle *strikten* Binärbäume (d.h. ein Knoten hat entweder keinen oder zwei Nachfolger), in denen unendlich tiefe Verzweigung zulässig ist während jeder Knoten mit einem Element von `List a` $= A^*$ beschriftet ist.
Stellen wir uns vor, zwei Personen laufen unabhängig voneinander durch zwei Terrains (die gleich sein können), und können miteinander kommunizieren. Sie tauschen sich darüber aus, was sie sehen, und wohin sie als nächstes gehen (links oder rechts). Auf ihrem Weg durch das Terrain könnte es nun passieren, dass sie an einem Punkt verschiedene Dinge (`objects`) sehen; in dem Fall befanden sie sich offensichtlich nicht am „gleichen“ Startpunkt. Oder aber sie führen die Prozedur unendlich lange fort und stellen keine Unterschiede fest. In dem Fall kann man ihre beiden Startpunkte „gleich“ nennen.
Das ist das Prinzip der Bisimulation für diesen Datentyp.

Definition 4.48. $R \subseteq \mathit{Terrain}\, a \times \mathit{Terrain}\, a$ heisst ($\mathit{Terrain}\, a$-)Bisimulation, wenn für alle $x \,\mathrm{R}\, y$ gilt:

1. $\mathit{objects}(x) = \mathit{objects}(y)$,
2. falls $x@\mathit{stuck}$, dann gilt auch $y@\mathit{stuck}$ (und $\mathit{deadend}(x) = \mathit{deadend}(y)$),
3. falls $x@\mathit{junction}$, dann gilt auch $y@\mathit{junction}$ und außerdem:
   (a) $(\mathit{left}\, x)\,\mathrm{R}(\mathit{left}\, y)$
   (b) $(\mathit{right}\, x)\,\mathrm{R}(\mathit{right}\, y)$.

Satz 4.49. *Wenn* R *eine* $\mathit{Terrain}\, a$*-Bisimulation ist und* $x \,\mathrm{R}\, y$*, dann* $x = y$*.*

Beweis. Beweisidee: Zeige durch Induktion über die Höhe des Baumes d: Falls $x \,\mathrm{R}\, y$, dann können wir in der finalen Koalgebra die Knoten bis zur Höhe d nicht voneinander unterscheiden. □

Fakt 4.50. $\mathit{Id}_{\mathit{Terrain}\, a} = \{(x, x) \mid x \in \mathit{Terrain}\, a\}$ *ist eine Bisimulation.*

Fakt 4.51. *Wenn* S *und* R *Bisimulationen sind, dann ist* $S \cup R$ *auch eine Bisimulation.*

Hieraus ergeben sich zwei einfache Erweiterungen des Prinzips des Beweisens per Koinduktion: Wenn wir wissen, dass S eine Bisimulation ist und wir zeigen wollen, dass $R \cup S$ auch eine Bisimulation ist, dann genügt es, die Bedingungen 1.-3. (siehe Definition 4.48) für alle Paare $(x, y) \in R \subseteq R \cup S$ zu zeigen. Weiterhin ist die Identität immer eine Bisimulation, so dass wir Bisimulationen grundsätzlich in der Form $R = \{\dots\} \cup \mathit{Id}$ definieren können. Um beispielsweise $(\mathit{left}\, x)\,\mathrm{R}(\mathit{left}\, y)$ zu zeigen, genügt es dann, $(\mathit{left}\, x) = (\mathit{left}\, y)$ zu zeigen.

Beispiel 4.52. Wir möchten nun eine korekursive Funktion *loop* definieren, die ein Terrain s auf das Terrain $\mathit{loop}\, s$ abbildet in dem alle Sackgassen durch den Ausgangspunkt von s ersetzt werden, z.B.:

$$\mathit{loop} : \mathit{Terrain}\, a \to \mathit{Terrain}\, a$$

$$\mathit{loop}(\quad) \mapsto$$

Hierzu haben wir die Idee, eine Hilfsfunktion *loopBack* zu verwenden, so dass $\mathit{loopBack}\, s\, t$ den Graph t so verändert, dass alle „stuck"-Knoten so aussehen und sich so verhalten, wie s. Voraussetzung hierfür ist, dass $s@\mathit{junction}$!

Wir nehmen also an, dass $s@junction$ und definieren *loop* und *loopBack* wie folgt:

$$loop\, s = loopBack\ \ s\ s$$

$$objects(loopBack\ \ t@junction\ s) = objects\ \ t$$
$$objects(loopBack\ \ t@stuck\ s) = objects\ \ s$$

$$left(loopBack\ \ t@junction\ s) = loopBack\ (left\, t)\ s$$
$$left(loopBack\ \ t@stuck\ s) = loopBack\ (left\, s)\ s$$

$$right(loopBack\ \ t@junction\ s) = loopBack\ (right\, t)\ s$$
$$right(loopBack\ \ t@stuck\ s) = loopBack\ (right\, s)\ s$$

Nun möchten wir die folgende Eigenschaft beweisen:

$$\forall s \in Terrain\, a@junction.\, loop(loop\ \ s) = loop\ \ s$$

Wir beweisen diese Eigenschaft per Koinduktion, d.h. wir definieren eine geeignete Relation R und zeigen, dass R eine $Terrain\, a$-Bisimulation ist. Hierzu definieren wir zunächst:

$$\mathrm{R} = \{\ (loop(loop\ \ s),\ \ loop\ \ s) \mid s \in Terrain\, a@junction\}.$$

Wir wollen nun zeigen, dass R eine Bisimulation ist und betrachten die Paare $loop(loop\ \ s)$ R $\ loop\ \ s$ mit $s \in Terrain\, a@junction$:

1. Wir beginnen mit der linken Seite des Paares.

$$\begin{aligned} objects(loop(loop\ \ s)) &= objects(loopBack\ (\ \underbrace{loop\ \ s}_{@junction}\ )\ (loop\ \ s)) \\ &= objects(loop\ \ s) \end{aligned}$$

2. Offensichtlich ist $(loop(loop\ \ s))@junction$, also ist dieser Fall trivial.
3. Offensichtlich ist $(loop\ \ s)@junction$.

   (a) Wir möchten zeigen, dass $left(loop(loop\, s))$ R $\ left(loop\, s)$ und betrachten hierzu zunächst wieder die linke Seite des Paares:

$$\begin{aligned} left(loop(loop\ \ s)) &= left(loopBack\ (loop\ \ s)\ (loop\ \ s)) \\ &= loopBack\ (left(loop\ \ s))\ (loop\ \ s) \\ &= loopBack\ (left(loopBack\ \ s\ s))\ (loop\ \ s) \\ &= loopBack\ (loopBack(left\ \ s\ s))\ (loop\ \ s) \end{aligned}$$

An diesem Punkt erweitern wir die ursprüngliche Relation R und setzen

$$\mathrm{R}' = \mathrm{R} \cup \{ \ (\mathit{loopBack}\ (\mathit{loopBack}\ \ t\ s)\ (\mathit{loop}\ \ s),\ \ \mathit{loopBack}\ \ t\ s) \mid \\ t \in \mathit{Terrain}\, a, s \in \mathit{Terrain}\, a@\mathit{junction}\}.$$

Nun gilt $\mathit{loopBack}(\mathit{loopBack}\ (\mathit{left}\ \ s)\ s)\ (\mathit{loop}\, s)\ \ \mathrm{R}'\ \ \mathit{loopBack}\ (\mathit{left}\ \ s)\ s$. Für die rechte Seite des Paares erhalten wir

$$\begin{aligned} \mathit{left}(\mathit{loop}\ \ s) &= \mathit{left}(\mathit{loopBack}\ \ s\ s) \\ &= \mathit{loopBack}\ (\mathit{left}\ \ s)\ s, \end{aligned}$$

was diesen Fall beendet.

(b) analog zu a) mit $\mathit{right}(\dots)$ anstelle von $\mathit{left}(\dots)$.

Es bleiben nun noch die Paare $\mathit{loopBack}\ (\mathit{loopBack}\ \ t\ s)\ (\mathit{loop}\ \ s)\ \ \mathrm{R}'\ \ \mathit{loopBack}\ \ t\ s$ mit $t \in \mathit{Terrain}\, a$ und $s \in \mathit{Terrain}\, a@\mathit{junction}$ zu betrachten:

1. Es gilt

$$\mathit{objects}(\mathit{loopBack}\ (\mathit{loopBack}\ \ t\ s)\ (\mathit{loop}\ \ s)) = \mathit{objects}(\mathit{loopBack}\, t\ s).$$

2. Da $\mathit{loopBack}\ (\mathit{loopBack}\ t\ s)\ (\mathit{loop}\ \ s)@\mathit{junction}$ und $\mathit{loopBack}\ \ t\ s@\mathit{junction}$, ist dieser Fall erneut trivial.

3. (a) Wir haben

$$\begin{aligned} &\mathit{left}(\mathit{loopBack}\ (\mathit{loopBack}\ \ t\ s)\ (\mathit{loop}\ \ s)) = \\ &\mathit{loopBack}\ (\mathit{left}(\mathit{loopBack}\ \ t\ s))\ (\mathit{loop}\ \ s) = \\ &\begin{cases} \mathit{loopBack}\ (\mathit{loopBack}\ (\mathit{left}\ \ s)\ s)\ (\mathit{loop}\ \ s) & \text{falls } t@\mathit{stuck} \\ \mathit{loopBack}\ (\mathit{loopBack}\ (\mathit{left}\ \ t)\ s)\ (\mathit{loop}\ \ s) & \text{falls } t@\mathit{junction} \end{cases} \end{aligned}$$

und

$$\mathit{left}(\mathit{loopBack}\ \ t\ s) = \begin{cases} \mathit{loopBack}\ (\mathit{left}\, s)\ s & \text{falls } t@\mathit{stuck} \\ \mathit{loopBack}\ (\mathit{left}\, t)\ s & \text{falls } t@\mathit{junction}, \end{cases}$$

so dass die Terme für die linke und die rechte Seite des Paares in beiden Fällen in der Relation R' enthalten sind.

(b) erneut analog zu a) mit $\mathit{right}(\dots)$ anstelle von $\mathit{left}(\dots)$.

Beispiel 4.53. Im Folgenden wollen wir einen Kodatentyp *InfTerrain* definieren, der sowohl vom bereits bekannten Typparameter a abhängt, als auch von einem zusätzlichen „Richtungs-Parameter“ d:

```
1 codata InfTerrain d a
2         objects: InfTerrain d a -> List a
3         move: InfTerrain d a -> (d -> InfTerrain d a)
```

Dies definiert die finale Koalgebra zu dem Mengenoperator

$$GX = \underbrace{A^*}_{objects} \times \underbrace{X^D}_{move} = A^* \times (D \to X),$$

wobei $A = [\![a]\!]$ und $D = [\![d]\!]$.

Eine Relation R ist dann eine *InfTerrain d a*-Bisimulation, wenn für alle $x \,\mathrm{R}\, y$ gilt:

1. $objects\, x = objects\, y$
2. $\forall dir \in d : (move\ \ x\ \ dir)\,\mathrm{R}(move\ \ y\ \ dir)$.

5 Polymorphie und System F

Unter *Polymorphie* versteht man die Anwendung syntaktisch gleicher Operatoren auf verschiedene Typen. Ein einfaches Beispiel ist die Überladung von Operatoren; so kann man etwa in C den operator + u.a. auf Integers, Floats oder Strings anwenden. In Java kann man überladene Operatoren mittels Interfaces selbst implementieren, wie in folgendem Beispiel:

```
1  interface Figure {
2          public void draw();
3  }
4
5  class Circle implements Figure {...};
6  class Triangle implements Figure {...};
7
8  public void drawAll (Figure [] figs) {
9          for (Figure f : figs)
10                 f.draw();
11 }
```

Ähnlich funktioniert der Typklassenmechanismus in Haskell, z.B. in

```
1  class Eq a where
2    (==) :: a -> a -> Bool
3    (/=) :: a -> a -> Bool
4
5  instance Eq Bool where
6      (==) True  True  = True
7      (==) False False = True
8      (==) _     _     = False
9      ...
```

Eine gänzlich andere Form von Polymorphie finden wir in folgendem Programm:

```
1  data List a = Nil | Cons a (List a)
2          append :: List a -> List a -> List a
3          append Nil ys = ys
4          append (Cons x xs) ys = Cons x (append xs ys)
```

Die zwei Formen von Polymorphie unterscheiden wir wie folgt:

- *Ad-hoc Polymorphie*: Hier tragen die verschiedenen Implementierungen einer Operation nur den gleichen Namen; jede Instanz kann sich anders verhalten. Man kann die Namensgleichheit aber für generische Programme wie die Methode *drawAll* im obigen Beispiel nutzen.

  Beispiele: Operatorenüberladung in C++, Methodenüberschreibung/Interfaces in C++ oder Java wie oben, Typklassen in Haskell.

- *Parametrische Polymorphie*: Eine einzelne Codepassage erhält einen generischen Typ, d.h. das Verhalten der Funktion ist *gleichförmig* auf allen Instanzen.

  Beispiele: Haskell, Java Generics (letzteres cum grano salis wegen des `instanceof`-Operators, der eine typabhängige Implementierung ermöglicht).

Ad-Hoc-Polymorphie trägt bei hinreichender Ausdrucksmächtigkeit (z.B. Typklassen) zur besseren Lesbarkeit und Wartbarkeit des Codes bei, ist aber in erste Linie ein syntaktisches Feature, das keine besonderen semantischen Änderungen nach sich zieht und syntaktisch herauskodiert werden kann. Parametrische Polymorphie ist weitaus fundamentaler und mächtiger, insbesondere dann, wenn man, wie z.B. in Haskell (mit *Glasgow extensions*), auch Funktionen zulässt, die polymorphe Funktionen als Argumente erwarten (*higher-rank polymorphism*). Wir befassen uns im folgenden mit Polymorphie in dieser letzteren Form, und führen dazu ein polymorphes Typsystem für den λ-Kalkül ein.

Wir erinnern zunächst an die Typisierungsregeln des einfach getypten λ-Kalküls $\lambda \to$:

$$1.\quad (\text{Axiom})\ \frac{}{\Gamma,\ x : \alpha\ \vdash\ x : \alpha}$$

$$2.\quad (\to_i)\ \frac{\Gamma,\ x : \alpha\ \vdash\ s : \beta}{\Gamma\ \vdash\ \lambda x.s : \alpha \to \beta}$$

$$3.\quad (\to_e)\ \frac{\Gamma\ \vdash\ s : \alpha \to \beta \qquad \Gamma\ \vdash\ t : \alpha}{\Gamma\ \vdash\ st : \beta}$$

Hier ein Beispiel, in dem ein Ausdruck mehrere Instanzen einer polymorphen Funktion verwendet:

```
1 silly :: Int
2 silly = (\x y -> x) (id True) (id 42)
```

Können wir `silly` einen Typ im einfach getypten λ-Kalkül zuweisen? Wir starten einen Versuch im Kontext

$$\Gamma = \{\mathit{True} : \mathit{Bool}, \mathit{False} : \mathit{Bool}, 0 : \mathit{Int}, 1 : \mathit{Int}, \ldots, 42 : \mathit{Int}, \ldots, \mathit{id} : a \to a\}$$

$\lambda x.x : a \to a$

$$\dfrac{\dfrac{\Gamma \vdash id : Bool \to Bool??}{\Gamma \vdash (id\ True) : Bool} \quad \dfrac{\Gamma \vdash id : Int \to Int??}{\Gamma \vdash (id\ 42) : Int}}{\dfrac{\vdots \qquad\qquad \vdots}{\Gamma \vdash (\lambda xy.\, x)\ (id\ True)\ (id\ 42) : Int}}$$

Die beiden Blätter des Ableitungsbaums sind in $\lambda \to$ nicht herleitbar, da nach der Axiomenregel Variablen nur genau den Typ bekommen, den sie im Kontext haben (und das kann wiederum nach der Definition von Kontexten nur einer sein, d.h. Änderung des Kontexts auf z.B. $id : Int \to Int$ löst das Problem nicht). Wir müssen das Typsystem also schon für diesen relativ harmlosen Fall erweitern. Wir bleiben zunächst beim Curry-Stil, d.h. bei Termen ohne Typannotationen. Wir warnen an dieser Stelle vor, dass der Unterschied zwischen der Church- und der Curry-Variante für System F, anders als für $\lambda \to$, durchaus wesentlich ist.

Definition 5.1 (System-F nach Curry)**.** System F nach Curry (oder $\lambda 2$-Curry) hat die gleiche Termsprache wie der ungetypte λ-Kalkül (und $\lambda\to$-Curry), also $s, t ::= x \mid st \mid \lambda x.s$. *Typen* $\alpha, \beta, \ldots$ in System F werden durch die Grammatik

$$\alpha, \beta ::= a \mid \alpha \to \beta \mid \forall a.\, \alpha \qquad (a \in \mathbf{B})$$

definiert. Der Typ $\forall a.\, \alpha$ ist zu lesen als der Typ der in a polymorphen Objekte von Typ α. Wir folgen weiter der Konvention, dass Bindungsoperatoren (wie $\forall$) einen möglichst weit nach rechts reichenden Scope bekommen. Mit $FV(\alpha)$ bzw. $FV(\Gamma)$ bezeichnen wir die *freien Typvariablen* eines Typs α bzw. eines Kontexts Γ, d.h. diejenigen, die nicht durch einen Quantor $\forall$ gebunden sind. Die Typisierung $\Gamma \vdash t : \alpha$ von Termen in Kontext wird induktiv durch die Typregeln von $\lambda\to$ sowie

- $(\forall_i)\ \dfrac{\Gamma \ \vdash \ s : \alpha \qquad a \notin FV(\Gamma)}{\Gamma \ \vdash \ s : \forall a.\alpha}$

- $(\forall_e)\ \dfrac{\Gamma \ \vdash \ s : \forall a.\alpha}{\Gamma \ \vdash \ s : (\alpha[a := \beta])}$

definiert.

Beispiel 5.2. Im System F lässt sich `silly` typisieren, wenn wir im Kontext Γ statt $id : a \to a$ einen explizit polymorphen Typ $id : \forall a.\, a \to a$ angeben. Wir können dann z.B. das linke noch offene Blatt in unserer obigen unvollständigen Herleitung in System F wie folgt herleiten:

$$(\forall_a)\ \dfrac{\Gamma \vdash id : \forall a.\, a \to a}{\Gamma \vdash id : Bool \to Bool}$$

5.0.1 Church-Kodierung in System F

In System F können wir die Kodierung von Datentypen, insbesonder die *Church-Kodierung* der natürlichen Zahlen, typisieren; im einzelnen:

Natürliche Zahlen:

- $\mathbb{N} := \forall a.\, (a \to a) \to a \to a$
- $zero : \mathbb{N}$
  $zero : \lambda f a.\, a$
- $succ : \mathbb{N} \to \mathbb{N}$
  $succ = \lambda n f a.\, f\ (n\ f\ a)$
- $fold : \forall a.(a \to a) \to a \to \mathbb{N} \to a$
  $fold = \lambda f x n.\, n\ f\ x$
- $add : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
  $add = \lambda n.\, fold\ succ\ n$

Paare:

- $(a \times b) := \forall r.\, (a \to b \to r) \to r$
- $pair : \forall a b.\, a \to b \to (a \times b)$
  $pair = \lambda x y f.\, f\ x\ y$
- $fst : \forall a b.\, (a \times b) \to a$
  $fst = \lambda p.\, p\ (\lambda x y.\, x)$
- $snd : \forall a b.\, (a \times b) \to b$
  $snd = \lambda p.\, p\ (\lambda x y.\, y)$

Summen:

- $(a + b) := \forall r.\, (a \to r) \to (b \to r) \to r$
- $inl : \forall a b.\, a \to (a + b)$
  $inl = \lambda \overset{a}{\downarrow} x f g.\, f\ x$
- $inr : \forall a b.\, b \to (a + b)$
  $inr = \lambda \overset{b}{\downarrow} y f g.\, g\ y$
- $case : \forall a b s.\, (a \to s) \to (b \to s) \to (a + b) \to s$
  $case = \lambda f g s.\, s\ f\ g$

```
1 data Either a b where
2         inl: a -> Either a b
3         inr: b -> Either a b
4
5 f: Either Int Bool -> Int
6 f (inl n) = n                          -- Typ: Int -> Int
7 f (inr b) = if b then 1 else 0         -- Typ: Bool -> Int
```

Listen:

- $List\ a := \forall r.\, r \to (a \to r \to r) \to r$
- $Nil : \forall a.\, List\ a$
  $Nil = \lambda u f.\, u$
- $Cons : \forall a.\, a \to List\ a \to List\ a$
  $Cons = \lambda x l u f.\, f\ x\ (l\ u\ f)$
- $len : \forall a.\, List\ a \to \mathbb{N}$
  $len = \lambda l.\, l\ zero\ (\lambda x r.\, succ\ r)$

Als Beispiel hier die Herleitung der Typisierung von *succ*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash n : \forall a.(a \to a) \to a \to a}(\forall_e)}{\Gamma \vdash n : (a \to a) \to a \to a} \quad \dfrac{}{\Gamma \vdash f : a \to a}}{\Gamma \vdash n\ f : a \to a}(\to_e) \quad \dfrac{}{\Gamma \vdash x : a}}{\boxed{\Gamma \vdash n\ f\ x : a}}(\to_e) \quad \begin{array}{c}\vdots\\ \boxed{\Gamma \vdash n\ f\ x : a}\end{array}}{\Gamma \vdash f\ (n\ f\ x) : a}(\to_e) \quad \dfrac{}{\Gamma \vdash f : a \to a}}{n : Nat \vdash \lambda f\, x\, .\, f\ (n\ f\ x) : (a \to a) \to a \to a}(\to_i) \times 2}{n : Nat \vdash \lambda f\, x\, .\, f\ (n\ f\ x) : Nat}(\forall_i)}{\vdash \lambda n\, f\, x\, .\, f\ (n\ f\ x) : Nat \to Nat}(\to_i)$$

Eine durchaus erwartete Eigenschaft ist die Bewahrung von Typisierung durch Reduktion, wie wir sie bereits für $\lambda\!\to$ gezeigt haben:

Satz 5.3 (Subject Reduction)**.** *Wenn* $\Gamma \vdash s : \alpha$ *und* $s \to_\beta t$*, dann* $\Gamma \vdash t : \alpha$

Wesentlich erstaunlicher ist dagegen folgende Aussage:

Satz 5.4 (Normalisierung, Girard)**.** $\lambda 2$ *ist stark normalisierend; d.h. wenn Wenn* $\Gamma \vdash s : \alpha$ *in* $\lambda 2$*, dann ist* s *stark normalisierend.*

Wir verschieben den nichttrivialen Beweis auf Abschnitt 5.3.

Tatsächlich kann jede totale berechenbare Funktion, die in Arithmetik zweiter Stufe definierbar ist, als Term in System F geschrieben werden! Dies schließt ein

- Alle primitiv rekursiven Funktionen
- Die Ackermannfunktion (die bekanntermaßen nicht primitiv rekursiv ist)
- Intuitiv: Compiler sind definierbar, Interpreter nicht (warum letzteres?).

Typinferenz und sogar Typüberprüfung im System F sind *unentscheidbar* (Wells, 1994). Zwei Ansätze zur Lösung:

- Einschränken des Typsystems zur sogenannten ML-Polymorphie (in der dann weniger Terme typisierbar sind)
- Übergang zu einem Church-System.

5.1 Curry vs. Church

Wir erinnern daran, dass sich Church-Systeme des λ-Kalküls von Curry-Systemen dadurch unterscheiden, dass sie eine explizite Typisierung von Variablen in λ-Abstraktionen verlangen; die Church-Variante von $\lambda\!\!\rightarrow$ z.B. hat durch die Grammatik

$$t, s ::= x \mid t\ s \mid \lambda \underline{x : \alpha}.t$$

definierte Terme, und die Typregel $(\rightarrow_i)$ hat die Form

$$\frac{\Gamma,\, x : \alpha \ \vdash\ t : \beta}{\Gamma \ \vdash\ \lambda x : \alpha \,.\, t \ : \alpha \rightarrow \beta}$$

– d.h. bei Rückwärtsanwendung der Typregeln muss man Typen der Variablen im Kontext nie erfinden, da sie ausdrücklich im Term deklariert sind.

Die Church-Variante von System F (oder $\lambda 2$-*Church*) hat dementsprechend durch die Grammatik

$$t, s ::= x \mid t\ s \mid \lambda x : a.t \mid \Lambda a.t \mid t\ \alpha$$

definierte Terme. Hierbei ist $\Lambda a.\, t$ ein über a explizit polymorpher Term, und Anwendung $t\ \alpha$ eines polymorphen Terms t auf einen Typ α liefert die Instanz von t für den Typ α. Hierfür haben wir eine eigene Form von β-Reduktion:

$$\Lambda a.t)\alpha \rightarrow_\beta t[a \mapsto \alpha].$$

Die Typregeln des Systems sind

- $(\forall_e)\ \dfrac{\Gamma \ \vdash\ t : \forall a.\alpha}{\Gamma \ \vdash\ t\beta : \alpha[a \mapsto \beta]}$

- $(\forall_i)\ \dfrac{\Gamma \ \vdash\ t : \alpha}{\Gamma \ \vdash\ \Lambda a.t : \forall a.\alpha}$

Beispiel 5.5.

1. $\vdash \Lambda a.\, \lambda x : a.\, x : \forall a.\, a \rightarrow a$
2. $\vdash \lambda x : \mathbb{N}.\, succ\ (id\ \mathbb{N}\ x) : \mathbb{N} \rightarrow \mathbb{N}$
3. $\vdash \Lambda b.\, \lambda x : (\forall a.\, a).\, x\ b : \forall b.\, (\forall a.\, a) \rightarrow b$
4. $\vdash \Lambda b.\, \lambda x : (\forall a.\, a).\, x\ ((\forall a.a) \rightarrow b)\ x : \forall b.(\forall a.a) \rightarrow b$

Die beiden letzten Beispiele demonstrieren das Prinzip *ex falso quodlibet*, d.h. aus einer falschen Annahme kann Beliebiges gefolgert werden. Hierbei spielt der Typ $\forall a.\, a$ die Rolle des *falsum*, das somit als äquivalent zu „jede Aussage ist wahr“ definiert ist.

5.2 ML-Polymorphie

ML-Polymorphie ist eine Einschränkung von System F á la Curry, die vielen funktionalen Programmiersprachen zu Grunde liegt, u.a. eben ML, aber auch der Grundversion von Haskell (die Glasgow Extensions verwenden volles System F und mehr). In ML-Polymorphie ist Typinferenz entscheidbar, durch einen ganz ähnlichen Algorithmus, wie wir ihn schon für $\lambda\to$ verwendet haben. Im einzelnen haben wir folgende Anpassungen:

- $\forall$ ist nur auf oberster Ebene von Typausdrücken erlaubt (d.h. es darf nicht im linken Argument von $\to$ auftauchen).

  Beispielsweise der Typ $(\forall a.\, a) \to b$ aus obigem Beispiel nicht mehr erlaubt, sehr wohl dagegen der Typ der Identität, $\forall a.\, a \to a$.

- Mehrfachinstanziierung polymorpher Funktionen ist nur per *let* erlaubt (daher wird ML-Polymorphie auch als *let-Polymorphie* bezeichnet):

$$\text{let } id = \lambda x.x \text{ in } \underbrace{id}_{(a\to a)\to a\to a}\ (\underbrace{id}_{a\to a})$$

Formal definieren wir den *ML-polymorphen λ-Kalkül* wie folgt. Wir unterscheiden zwischen *Typen*, definiert durch die Grammatik

$$\alpha, \beta ::= a \mid \alpha \to \beta,$$

und *Typschemata*, gegeben durch die nicht-rekursive Grammatik

$$S ::= \forall a_1, \ldots, a_k : \alpha \qquad (k \geq 0).$$

(Insbesondere sind Typen Typschemata, aber nicht umgekehrt.) Für Terme führen wir zusätzlich das let-Konstrukt ein, d.h. Terme sind gegeben durch die Grammatik

$$t, s ::= x \mid t\, s \mid \lambda x.t \mid \text{let } x = t \text{ in } s.$$

Kontexte haben die Form $\Gamma = (x_1 : S_1, \ldots, x_n : S_n)$ mit paarweise verschiedenen Variablen x_i, d.h. weisen Variablen Typschemata zu. Typisierungsurteile sind weiterhin von der Form $\Gamma \vdash t : \alpha$, wobei wir Typvariablen, die im Typ α, aber nicht im Kontext Γ vorkommen, als implizit allquantifiziert lesen, d.h. für $FV(\alpha) \setminus FV(\Gamma) = \{a_1, \ldots, a_k\}$ setzen wir

$$Cl(\Gamma, \alpha) = \forall a_1.\ \ldots \forall a_k.\, \alpha.$$

Z.B. haben wir wie bisher $\vdash\ \lambda x.x : a \to a$, und nach obiger Definition gilt $Cl\Big((), a \to a\Big) = \forall a.a \to a$.

Die Typisierungsregeln sind wie folgt:

- $(\forall_e)\ \dfrac{}{\Gamma\ \vdash\ x : \alpha[a_1 \mapsto \beta_1, \ldots, a_k \mapsto \beta_k]} \quad (x : \forall a_1. \ldots \forall a_k.\, \alpha) \in \Gamma$

- $(\to_i)\ \dfrac{\Gamma,\, x:\alpha \ \vdash\ t:\beta}{\Gamma \ \vdash\ \lambda x.t:\alpha\to\beta}$

- $(\to_e)\ \dfrac{\Gamma \ \vdash\ t:\alpha\to\beta \qquad \Gamma \ \vdash\ s:\alpha}{\Gamma \ \vdash\ ts:\beta}$

- $(\text{let})\ \dfrac{\Gamma \ \vdash\ t:\alpha \qquad \Gamma,\, x:Cl(\Gamma,\alpha) \ \vdash\ s:\beta}{\Gamma \ \vdash\ \text{let}\, x=t \text{ in } s:\beta}$

Beispiel 5.6. Unseren Beispielterm von oben können wir in diesem System typisieren. Eine naive Suche nach einer Typherleitung verläuft etwa wie folgt:

$$\dfrac{\dfrac{\dots}{\vdash \lambda x.\, x : a\to a} \qquad \dfrac{id : \forall a.\, a\to a \ \vdash\ id:\gamma\to\beta \qquad id:\forall a.\, a\to a \ \vdash\ id:\gamma}{id:\forall a.\, a\to a \ \vdash\ id\ id:\beta}}{\vdash\ \text{let}\, id = \lambda x.\, x \text{ in } id\ id : \beta}$$

Wir können dann z.B. $\gamma = b\to b$, $\beta = b\to b$ wählen.

Achtung: Intuitiv denkt man, dass man $\text{let}\, id = \lambda x.\, x \text{ in } id\ id$ kürzer auch als $(\lambda id.\, id\ id)\ \lambda x.\, x$ schreiben kann. Der geklammerte Ausdruck ist aber nicht typisierbar, da polymorphe Verwendung von Variablen nur dann vorgesehen ist, wenn diese durch let gebunden sind.

Typinferenz unter ML-Polymorphie ist wie versprochen durch eine Erweiterung unseres Algorithmus für $\lambda\to$ entscheidbar:

Wir haben wieder ein Inversionslemma, das für λ-Abstraktion und Applikation wie für $\lambda\to$ lautet; für Variablen haben wir leicht verallgemeinert

- Wenn $\Gamma \vdash x : \gamma$, dann existieren Typen β_i, und ein Typschema $S = \forall a_1 \dots \forall a_k.\, \alpha$, so dass $(x:S)\in\Gamma$ und $\gamma = \alpha[\beta_1/a_1, \dots, \beta_k/a_k]$.

Für das neue let-Konstrukt schließlich haben wir

- Wenn $\Gamma \vdash (\text{let}\ x = s\ \text{in}\ t) : \alpha$, dann existiert ein Typ β mit $\Gamma \vdash s : \beta$ und $\Gamma, x : Cl(\Gamma,\beta) \vdash t : \alpha$.

(Der Beweis ist jeweils ebenso leicht wie im Fall von $\lambda\to$.) Hieraus ergibt sich unmittelbar folgende Anpassung von Algorithmus W (durch die er tatsächlich erst zum Algorithmus W wird, der von Damas und Milner (1982) bei seiner Einführung bereits für den ML-polymorphen λ-Kalkül formuliert wurde.). Wir erinnern daran, dass $PT(\Gamma; t; \alpha)$ ein Gleichungssystem auf Typen liefert, dessen allgemeinste Lösung σ gleichzeitig die allgemeinste Lösung von $\Gamma\sigma \vdash t : \alpha\sigma$ ist. Wir behalten in der rekursiven Definition die Klauseln für Applikation und λ-Abstraktion bei. Die Klausel für Variablen verallgemeinern wir für $(x : \forall a_1 \dots \forall a_k.\, \alpha) \in \Gamma$ zu

$$PT(\Gamma; x; \gamma) = \{\gamma \doteq \alpha[a'_1/a_1, \dots, a'_k/a_k]\}$$

für *frische* Variablen a_i'; damit stellen wir sicher, dass wir die a_i (jetzt a_i') bei verschiedenen Vorkommen von x verschieden instanziieren können. Für das neue let-Konstrukt haben wir das Problem, dass die Typvariable, den wir als Typ für s in let $x = s$ in t einführen, keine Berechnung von $Cl(\Gamma, \cdot)$ erlauben würde. Wir müssen das Teilproblem für s also zunächst tatsächlich lösen, d.h. die zugehörige Gleichungsmenge unifizieren. Damit ergibt sich

$$PT(\Gamma; (\text{let } x = s \text{ in } t); \alpha) = PT(\Gamma\sigma, x : Cl(\Gamma\sigma, \sigma(b); t; \alpha\sigma),$$

wobei

$$\sigma = mgu(PT(\Gamma; s; b))$$

mit b frisch.

Beispiel 5.7. Wir führen die Typinferenz für den oben bereits typisierten Term let $id = \lambda x.\, x$ in $id\ id$ durch (im leeren Kontext ()): Wir berechnen wie in $\lambda\!\!\rightarrow$

$$mgu(PT((); \lambda x.\, x; b) = [a \to a/b].$$

Wir berechnen dann gemäß der Klausel für let mit $\Gamma = (id : \forall a.\, a \to a)$

$$\begin{aligned} PT(\Gamma; id\ id; b) &= PT(\Gamma; id; c \to b) \cup PT(\Gamma; id; c) \\ &= \{c \to b \doteq a' \to a'\} \cup \{c \doteq a'' \to a''\}, \end{aligned}$$

mit mgu $[a'' \to a''/c, (a'' \to a'') \to a'' \to a''/b]$.

5.3 Starke Normalisierung in System F

Wir holen nunmehr den aufgeschobenen Beweis des Normalisierungssatzes für System F ($\lambda 2$) nach, d.h. wir zeigen, dass jeder im System typisierbare Term stark normalisierend ist. Der Beweis verallgemeinert den, den wir weiter oben für $\lambda\!\!\rightarrow$ angegeben haben; wir geben ihn dennoch hier vollständig an, mit wörtlicher Wiederholung der gemeinsamen Teile.

Die Hauptidee am Beweis ist die Definition einer Semantik für Typen α als Teilmengen

$$[\![\alpha]\!]_\xi \subseteq SN := \{t \in \Lambda \mid t \text{ stark normalisierend}\},$$

wobei Λ die Menge aller λ-Terme ist und ξ eine *Typumgebung*

$$\xi : \mathbf{B} \to SAT,$$

die also jede Typvariable durch eine *saturierte* Menge von stark normalisierenden Termen im Sinne von Definition 5.9 unten interpretiert. Wenn wir Korrektheit des Typsystems bezüglich dieser Semantik zeigen, d.h wenn wir zeigen, dass jeder typisierbare Term tatsächlich zur Interpretation seines Typs gehört, ist das Resultat offenbar bewiesen. Wir geben die Semantik rekursiv an: Für $A, B \subseteq \Lambda$ schreiben wir

$$A \to B = \{t \in \Lambda \mid \forall s \in A.\, ts \in B\}$$

und setzen dann

$$\begin{aligned}[\![a]\!]_\xi &= \xi(a)\\ [\![\alpha \to \beta]\!]_\xi &= [\![\alpha]\!]_\xi \to [\![\beta]\!]_\xi\\ [\![\forall a.\, \alpha]\!]_\xi &= \bigcap_{A \in SAT} \alpha]\!]_{\xi[a \mapsto A]}.\end{aligned}$$

Lemma 5.8 (Substitutionslemma)**.** *Wir haben* $[\![\alpha[\beta/a]]\!]_\xi = [\![\alpha]\!]_{\xi[a \mapsto [\![\beta]\!]_\xi]}$.

Beweis. Induktion über α. □

Die noch ausstehende Definition von Saturiertheit lautet wie folgt.

Definition 5.9. Eine Teilmenge $A \subseteq SN$ heißt *saturiert*, wenn

1. $xt_1 \ldots t_n \in A$ für alle Variablen x und alle $t_1, \ldots, t_n \in SN$, $n \geq 0$.
2. $t[s/x]u_1 \ldots u_n \in A \Rightarrow (\lambda x.\, t)su_1 \ldots u_n \in A$ für alle $s \in SN$.

Wir setzen dann

$$SAT = \{A \subseteq \Lambda \mid A \text{ saturiert}\}.$$

Lemma 5.10 (Saturiertheitslemma)**.**

a) $SN \in SAT$.

b) $[\![\alpha]\!]_\xi \in SAT$ *für alle* α, ξ.

Beweis. a):

1. Sei x eine Variable und $t_1, \ldots, t_n \in SN$. Zu zeigen ist $xt_1 \ldots t_n \in SN$. Das ist aber klar: Jeder Redukt von $xt_1 \ldots t_n$ ist von der Form $xt'_1 \ldots t'_n$ mit $t_i \to_\beta^* t'_i$ (also $t'_i \in SN$), so dass man aus einer unendlichen Reduktionssequenz für $xt_1 \ldots t_n$ auch eine für eines der t_i gewinnen würde, im Widerspruch zu $t_i \in SN$.
2. Sei $s \in SN$ und $t[s/x]u_1 \ldots u_n \in SN$; dann gilt $t, u_1, \ldots, u_n \in SN$. Zu zeigen ist $v := (\lambda x.\, t)su_1 \ldots u_n \in SN$. Da $t, s, u_1, \ldots, u_n \in SN$, hat jede unendliche Reduktionssequenz von v die Form

   $$(\lambda x.\, t)su_1 \ldots u_n \overset{*}{\to}_\beta (\lambda x.\, t')s'u'_1 \ldots u'_n \to_\beta t'[s'/x]u'_1 \ldots u'_n \to_\beta \ldots$$

   Da aber $t[s/x]u_1 \ldots u_n \to_\beta^* t'[s'/x]u'_1 \ldots u'_n$ (eventuell per Reduktion von mehreren Vorkommen von s in $t[s/x]$ ähnlich wie im Beweis des Critical Pair Lemma), ist dies ein Widerspruch zu $t[s/x]u_1 \ldots u_n \in SN$.

b): Induktion über α.

1. Für eine Typvariable a gilt $[\![a]\!]_\xi = \xi(a) \in SAT$.
2. Seien $A := [\![\alpha]\!]_\xi, B := [\![\beta]\!]_\xi$ saturiert; zu zeigen ist, dass $A \to B$ saturiert ist.

(a) $A \to B \subseteq SN$: Sei $t \in A \to B$. Sei x eine Variable. Da A saturiert ist, gilt $x \in A$, somit $tx \in B$ per Definition von $A \to B$, also $tx \in SN$ und somit $t \in SN$.

(b) Seien $r_1, \ldots, r_n \in SN$; zu zeigen ist $xr_1 \ldots r_n \in A \to B$. Sei also $s \in A \subseteq SN$, zu zeigen ist dann $xr_1 \ldots r_n s \in B$. Da $s \in SN$, folgt dies aus Saturiertheit von B.

(c) Sei $s \in SN$ und $t[s/x]r_1 \ldots r_n \in A \to B$. Zu zeigen ist $(\lambda x.\, t)sr_1 \ldots r_n \in A \to B$. Sei also $v \in A$, zu zeigen ist dann $(\lambda x.\, t)sr_1 \ldots r_n v \in B$. Per Saturiertheit von B reicht dazu $t[s/x]sr_1 \ldots r_n v \in B$, was aus $v \in a$ und $t[s/x]r_1 \ldots r_n \in A \to B$ folgt.

3. Für den Fall $\forall a.\, \alpha$ reicht es anzumerken, dass SAT offensichtlich abgeschlossen unter großen Durchschnitten nichtleerer Mengensysteme ist; aus Behauptung a) des Lemmas folgt, dass im Durchschnitt $\bigcap_{A \in SAT}$ mindestens ein Index A, nämlich $A = SN$, vorkommt.

□

Definition 5.11 (Erfülltheit/Konsequenz)**.** Sei σ eine Substitution und ξ eine Typumgebung. Dann schreiben wir

$$\begin{aligned}
\sigma, \xi \models t : \alpha &:\iff t\sigma \in [\![\alpha]\!]_\xi && (\sigma, \xi \textit{ erfüllen } t : \alpha) \\
\sigma, \xi \models \Gamma &:\iff \forall (x : \alpha) \in \Gamma.\, \sigma \models x : \alpha && (\sigma, \xi \textit{ erfüllen } \Gamma) \\
\Gamma \models t : \alpha &:\iff \forall \sigma, \xi.\, (\sigma, \xi \models \Gamma \Rightarrow \sigma, \xi \models (t : \alpha)) && (t : \alpha \text{ ist } \textit{Konsequenz} \text{ von } \Gamma)
\end{aligned}$$

Lemma 5.12 (Korrektheit)**.** *Wenn* $\Gamma \vdash t : \alpha$*, dann* $\Gamma \models t : \alpha$*.*

Diese Aussage hat den gleichen Charakter wie Korrektheitsaussagen über Beweissysteme: Wenn man aus Typisierungsannahmen Γ mittels der Typregeln herleiten kann, dass t Typ α hat, dann ist $t : \alpha$ auch eine Konsequenz aus Γ in der Semantik.

Beweis. Formal führen wir eine Induktion über die Herleitung von $\Gamma \vdash t : \alpha$ durch; informell heißt dies, dass wir zeigen, dass alle Regeln des Typsystems korrekt bezüglich der Semantik sind.

$$(\text{Ax})\ \frac{}{\Gamma \vdash x : \alpha}$$

Hier ist zu zeigen, dass $\Gamma \models x : \alpha$; das gilt trivialerweise.

$$(\to_e)\ \frac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash s : \alpha}{\Gamma \vdash ts : \beta}$$

Sei $\Gamma \models t : \alpha \to \beta$ und $\Gamma \models s : \alpha$. Zu zeigen ist dann $\Gamma \models ts : \beta$. Sei also $\sigma, \xi \models \Gamma$. Nach Annahme gilt $\sigma, \xi \models t : \alpha \to \beta$ und $\sigma, \xi \models s : \alpha$, d.h. $t\sigma \in [\![\alpha]\!]_\xi \to [\![\beta]\!]_\xi$ und $s\sigma \in [\![\alpha]\!]_\xi$, also $(ts)\sigma = (t\sigma)s\sigma \in [\![\beta]\!]_\xi$, d.h. $\sigma, \xi \models ts : \beta$.

$$(\to_i) \frac{\Gamma,\, x : \alpha \;\vdash\; t : \beta}{\Gamma \;\vdash\; \lambda x.\, t \;:\; \alpha \to \beta}$$

Sei $\Gamma,\, x : \alpha \;\models\; t : \beta$ und $\sigma, \xi \models \Gamma$. O.E. sei x frisch. Zu zeigen ist $(\lambda x.\, t)\sigma \in [\![\alpha]\!]_\xi \to [\![\beta]\!]_\xi$. Sei also $v \in [\![\alpha]\!]_\xi$; zu zeigen ist dann $((\lambda x.\, t)\sigma)v \in [\![\beta]\!]_\xi$. Dazu rechnen wir

$$\begin{array}{lll}
((\lambda x.\, t)\sigma)v \in [\![\beta]\!]_\xi & & \\
\Longleftarrow t\sigma[v/x] \in [\![\beta]\!]_\xi & & ([\![\beta]\!]_\xi \text{ saturiert}) \\
\iff \sigma[x \mapsto v], \xi \models t : \beta & & \\
\Longleftarrow \sigma[x \mapsto v], \xi \models \Gamma, x : \alpha & & (\text{Annahme}) \\
\iff \sigma[x \mapsto v], \xi \models x : \alpha & & (\sigma, \xi \models \Gamma) \\
\iff v \in [\![\alpha]\!]_\xi; & &
\end{array}$$

letzteres gilt nach Voraussetzung.

$$(\forall_e) \frac{\Gamma \vdash t : \forall a.\, \alpha}{\Gamma \vdash t : \alpha[\beta/a]}$$

Korrektheit dieser Regel ist klar, da

$$[\![\forall a.\, \alpha]\!]_\xi = \bigcap_{A \in SAT} [\![\alpha]\!]_{\xi[a \mapsto A]} \subseteq [\![\alpha]\!]_{\xi[a \mapsto [\![\beta]\!]_\xi]} = [\![\alpha[\beta/a]]\!]_\xi,$$

wobei wir im letzten Schritt das Substitutionslemma verwenden.

$$(\forall_i) \frac{\Gamma \vdash t : \alpha}{\Gamma \vdash t : \forall a.\, \alpha}\ (a \notin FV(\Gamma))$$

Sei $\Gamma \models t : \alpha$ und $\sigma, \xi \models \Gamma$. Da $a \notin FV(\Gamma)$ und somit $\xi(a)$ für die Erfülltheit von Γ keine Rolle spielt, gilt dann auch $\sigma, \xi[a \mapsto A] \models \Gamma$ für alle $A \in SAT$. Es folgt $\sigma, \xi[a \mapsto A] \models t : \alpha$ und somit $t\sigma \in [\![\alpha]\!]_{\xi[a \mapsto A]}$ für alle $A \in SAT$, d.h. $t\sigma \in [\![\forall a.\, \alpha]\!]_\xi$, also $\sigma, \xi \models t : \forall a.\, \alpha$. □

Daraus folgt im wesentlichen sofort unser Zielresultat:

Satz 5.13. *$\lambda 2$ ist stark normalisierend.*

Beweis. Sei $\Gamma \vdash t : \alpha$. Nach dem Korrektheitslemma folgt $\Gamma \models t : \alpha$. Setze $\sigma = [\,]$ und $\xi(a) = SN$ für alle a. Dann gilt $\sigma, \xi \models \Gamma$, da $x \in [\![\alpha]\!]_\xi$ für alle α per Saturiertheit von $[\![\alpha]\!]_\xi$. Es folgt $\sigma, \xi \models t : \alpha$, d.h. $t = t\sigma \in [\![\alpha]\!]_\xi \subseteq SN$. □

6 Reguläre Ausdrücke

6.1 Recall: Nichtdeterministische endliche Automaten

Definition 6.1. Ein *nichtdeterministischer endlicher Automat mit ϵ-Transitionen* (NFA^ϵ) ist ein Tupel

$$A = (Q, \Sigma, \Delta, s, F),$$

bestehend aus

- einer Menge Q von *Zuständen*;
- einem *Alphabet* Σ;
- einer *Transitionsabbildung* $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ (man erinnere sich, dass $\mathcal{P}(X)$ die *Potenzmenge* einer Menge X bezeichnet, also die Menge aller Teilmengen von X);
- einem *Anfangszustand* $s \in Q$;
- einer Menge $F \subseteq Q$ von *akzeptierenden* oder *finalen* Zuständen.

Wir schreiben

$$q \xrightarrow{a} q : \iff q' \in \Delta(q, a).$$

Definition 6.2. Definiere $q \overset{u}{\Longrightarrow} q'$ für Wörter $u \in \Sigma^*$ induktiv per

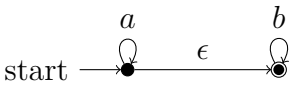
$$\frac{}{q \overset{\epsilon}{\Longrightarrow} q}$$

$$\frac{q \overset{u}{\Longrightarrow} q' \qquad q' \xrightarrow{\epsilon} q''}{q \overset{u}{\Longrightarrow} q''}$$

$$\frac{q \overset{u}{\Longrightarrow} q' \qquad q' \xrightarrow{a} q''}{q \overset{ua}{\Longrightarrow} q''}$$

Wir sagen dann, dass A ein Wort u *akzeptiert*, wenn ein akzeptierender Zustand $q \in F$ existiert mit $s \overset{u}{\Longrightarrow} q$. Eine *Sprache* ist naheliegenderweise eine Menge von Worten (über Σ). Die von A *akzeptierte Sprache* $L(A)$ ist dann die Menge aller von A akzeptierten Wörter. Eine Sprache $L \subseteq \Sigma^*$ heißt *regulär*, wenn ein NFA^ϵ A mit $L = L(A)$ existiert, d.h. wenn A L akzeptiert.

Beispiel 6.3. In Abbildungen von Automaten bezeichnen wir den initialen Zustand mit einem eingehenden Pfeil und finale Zustände durch einen zusätzlichen Kringel. Sei etwa A der NFA^ϵ



Dann haben wir

$$L(A) = \{a^n b^k \mid n, k \geq 0\}.$$

Definition 6.4. Ein NFA$^\epsilon$ A ist ein *nichtderterministischer endlicher Automat (NFA)* genau dann, wenn A keine ϵ-Transitionen hat, d.h. $\Delta(q, \epsilon) = \{\}$ für alle Zustände q.

Ein NFA A ist *deterministisch* (DFA), wenn $|\Delta(q, a)| = 1$ für alle Zustände q. Wir schreiben dann $\Delta(q, a) =: \{\delta(q, a)\}$

Hinsichtlich Ausdrucksstärke sind DFA, NFA und NFA$^\epsilon$ äquivalent:

Lemma 6.5 (Potenzmengenkonstruktion)**.** *Zu jedem NFA$^\epsilon$ A existiert ein DFA B mit $L(B) = L(A)$*

6.2 Reguläre Ausdrücke

Definition 6.6. *Reguläre Ausdrücke $r, s, \ldots$* sind durch die Grammatik

$$r, s ::= 1 \mid 0 \mid r + s \mid rs \mid r^* \mid a \qquad (a \in \Sigma)$$

definiert. Die Semantik eines regulären Ausdrucks r ist die wie folgt rekursiv definierte Sprache $L(r)$:

- $L(a) = \{a\}$
- $L(1) = \{\epsilon\}$
- $L(0) = \emptyset$
- $L(r + s) = L(r) \cup L(s)$
- $L(rs) = \{uv \mid u \in L(r), v \in L(s)\}$
- $L(r^*) = \{u_1 \ldots u_n \mid n \geq 0, u_i \in L(r) \forall i\}$

Beispiel 6.7. Wir können die aus allen Wörtern über $\Sigma = \{a, b\}$, die höchstens zwei aufeinanderfolgende a enthalten, durch den regulären Ausdruck

$$(b + ab + aab)^*(1 + a + aa)$$

(in `grep`-Notation: `(b|ab|aab)*(|a|aa)`) beschreiben.

Satz 6.8 (Satz von Kleene)**.** *Eine Sprache L ist genau dann regulär, wenn es einen regulären Ausdruck r gibt mit $L = L(r)$*

Beweis. „⇐“: Per Induktion über r:

- $r = 0$:

  start ⟶●

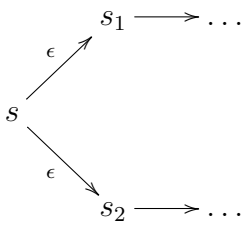
- $r = 1$:

  start $\longrightarrow$◉

- $r = a$:

  start $\longrightarrow$● $\xrightarrow{a}$ ◉

- $r = r_1 + r_2$: Nach IV haben wir Automaten A_i mit Anfangszuständen s_i, die $L(r_i)$ akzeptieren. Damit akzeptiert der Automat, der als disjunkte Vereinigung der A_i und hinzufügen eines neuen Anfangszustands s mit

  entsteht, offenbar $L(r_1) + L(r_2) = L(r_1 + r_2)$.

- $r = r_1 r_2$: Nach IV haben wir Automaten A_i mit Anfangszuständen s_i, die $L(r_i)$ akzeptieren. Wir konstruieren einen Automaten A ausgehend von der disjunkten Vereinigung von A_1 und A_2. Der initiale Zustand von A ist der initiale Zustand s_1 von A_1, und die akzeptierenden Zustände von A sind die von A_2. Für fügen ferner von jedem akzeptierenden Zustand von A_1 einenq ϵ-Übergang in den initialen Zustand von A_2 hinzu. Dann akzeptiert A gerade $L(r_1 r_2) = L(r_1)L(r_2)$: Um von A akzeptiert zu werden, muss ein Wort, ausgehend von s_1, einen finalen Zustand von A_2 erreichen, was nach Konstruktion gerade über einen finalen Zustand von A_1 und den initialen Zustand von A_2 möglich ist, so dass r aus einem von A_1 akzeptierten Wort gefolgt von einem von A_2 akzeptierten Wort besteht.

- für $r = r_0^*$: Nach IV haben wir einen NFA$^\epsilon$ A_0 mit $L(A) = r_0$. Wir erweitern A_0 zu einem NFA$^\epsilon$ A, indem wir einen neuen Zustand s als initialen Zustand hinzufügen, den wir gleichzeitig als einzigen finalen Zustand erklären, und ϵ-Transitionen von s in den initialen Zustand von A_0 sowie von den finalen Zuständen von A_0 nach s. Dann akzeptiert A gerade $L(r^*)$: Um von A akzeptiert zu werden, muss ein Wort u entweder leer sein oder akzeptiert werden, nachdem es einmal A_0 passiert hat und über einen finalen Zustand von A_0 wieder nach s gekommen ist, d.h. u muss akzeptiert werden, nachdem man ein nichtleeres Präfix aus $L(r)$ entfernt hat; per Induktion über die Länge von u folgt dann, dass der Rest von u in $L(r^*)$ liegt, also auch u.

„⇒“: Sei $A = (Q, \Sigma, \delta, s, F)$ ein deterministischer endlicher Automat. Wir zeigen

($*$): Für alle $q, q' \in Q$ und alle $R \subseteq Q$ gibt es einen regulären Ausdruck $r^R_{q,q'}$ mit

$$L(r^R_{q,q'}) = \{u \in \Sigma^* \mid q \xrightarrow{u} q' \text{ mit Zwischenzuständen nur aus } R\}.$$

Daraus folgt dann die Behauptung, denn $L(A) = \sum_{q \in F} r^Q_{s,q}$.

Beweis von $(*)$ per Induktion über $|R|$:

- Induktionsanfang ($|R| = 0$):
  - Fall 1: $q \neq q'$: $r^{\emptyset}_{q,q'} = \sum_{q \xrightarrow{a} q'} a$
  - Fall 2: $q = q'$: $r^{\emptyset}_{q,q'} = 1 + \sum_{q \xrightarrow{a} q'} a$
- Induktionsschritt: Sei $|R| > 0$. Wähle dann $q_0 \in R$, $R_0 := R \setminus \{q_0\}$. Wir haben nach Induktionsvoraussetzung schon alle $r^{R_0}_{q,q'}$ konstruiert. Wir setzen dann

$$r^R_{q,q'} := r^{R_0}_{q,q'} + r^{R_0}_{q,q_0} (r^{R_0}_{q_0,q_0})^* r^{R_0}_{q_0,q'}.$$

  Um zu sehen, dass $r^R_{q,q'}$ bei dieser Definition die in $(*)$ angegebene Sprache akzeptiert, gehen wir per Fallunterscheidung vor. Ein Weg per u von q nach q' mit Zwischenzuständen nur in R kann entweder q_0 passieren oder nicht. Letzterer Fall wird per Induktion gerade durch den linken Summanden $r^{R_0}_{q,q'}$ beschrieben, ersterer durch den rechten Summanden $r^{R_0}_{q,q_0}(r^{R_0}_{q_0,q_0})^* r^{R_0}_{q_0,q'}$: wir unterteilen den Weg jedes Mal, wenn er q_0 passiert, und konkatenieren die so entstehenden Teilwege, die jetzt nur noch durch Zwischenzustände in R_0 laufen.

□

Bemerkung 6.9. Aus dem Satz von Kleene folgt sofort, dass reguläre Ausdrücke unter Durchschnitt und Komplement von Sprachen abgeschlossen sind, da wir diese Konstruktionen auf DFA leicht realisieren können (wir komplementieren einen DFA, indem wir einfach die akzeptierenden Zustände zu nicht akzeptierenden machen und umgekehrt; Durchschnitte können wir in der bekannten Weise mittels Komplement und Vereinigung konstruieren: $r \cap s = \neg(\neg r + \neg s)$).

Wir merken an, dass Komplementierung von NFA nicht so einfach ist wie für DFA – wenn man z.B. die obige Konstruktion zur Komplementierung von DFAs auf einen NFA A für r anwendet, akzeptiert der so entstehende NFA B nicht unbedingt das Komplement von r: ein Wort $u \in L(r)$ kann in A durchaus auch nicht-finale Zustände erreichen (damit u von A akzeptiert wird, reicht es, dass u irgendeinen finalen Zustand erreicht, u kann aber auch andere Zustände erreichen), und würde dann von B akzeptiert. Um einen NFA zu komplementieren, wird man ihn daher typischerweise zunächst per Potenzmengenkonstruktion in einen DFA transformieren.

Durchschnittsbildung dagegen funktioniert ohne weiteres auch für NFA: Gegeben seien nichtdeterministische Automaten $A = (Q_A, \Sigma, \Delta_A, s_A, F_A)$ und $B = (Q_B, \Sigma, \Delta_B, s_B, F_B)$. Wir definieren dann einen *Produktautomaten*

$$A \times B := (Q_A \times Q_B, \Sigma, \Delta, (s_A, s_B), F_A \times F_B)$$

mit

$$\Delta(a, (p, q)) = \underbrace{\Delta_A(a, p)}_{\subseteq Q_A} \times \underbrace{\Delta(a, q)}_{\subseteq Q_B} \subseteq Q_A \times Q_B.$$

D.h. wir definieren finale Zustände und Transitionen in $A \times B$ durch

- (p, q) final $\Leftrightarrow$ p final und q final und
- $(p, q) \xrightarrow{a} (p', q') \Leftrightarrow p \xrightarrow{a} p'$ und $q \xrightarrow{a} q'$.

Man sieht leicht, dass $L(A \times B) = L(A) \cap L(B)$.

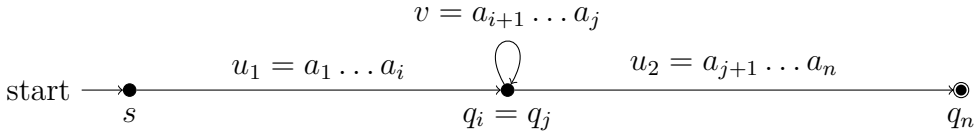
Satz 6.10 (Pumping-Lemma)**.** *Wenn eine Sprache L regulär ist, dann existiert $l \geq 1$, so dass für alle $w \in L$ mit $|w| \geq l$ eine Unterteilung $w = u_1 v u_2$ mit $|v| \geq 1$, $|u_1 v| \leq l$ existiert, so dass*

$$u_1 v^* u_2 \subseteq L.$$

Wir verwenden den Satz vor allem in kontraponierter Form:

> *Wenn für alle $l \geq 1$ ein $w \in L$ mit $|w| \geq l$ existiert, so dass*
>
> $$u_1 v^* u_2 \not\subseteq L$$
>
> *für alle Unterteilungen $w = u_1 v u_2$ mit $|v| \geq 1$ und $|u_1 v| \leq l$, dann ist L* nicht *regulär.*

Beweis (Satz 6.10). Sei $A = (Q, \Sigma, \delta, s, F)$ ein DFA mit $L(A) = L$. Setze dann $l = |Q|$. Sei $w \in L$ mit $|w| \geq l$, also $w = a_1 \dots a_n$ mit $n \geq l$. Dann $w \in L(A)$, also existiert $s = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ mit q_n final. Das sind mindestens $l + 1$ Zustände, also existieren $i < j \leq l$ mit $q_i = q_j$. Wir unterteilen u gerade bei i und j:



Nach Erreichen von q_i mittels u_1 können wir die Schleife bei $q_i = q_j$ mittels v beliebig oft durchlaufen (auch kein Mal) und erreichen dann letztlich mittels u_2 den finalen Zustand q_n, d.h. $u_1 v^* u_2 \subseteq L$ wie verlangt. □

Beispiel 6.11.

1. $L = \{a^n b^n \mid n \geq 0\}$ ist nicht regulär, denn:

   Sei $l \geq 1$ gegeben; wähle dann $w = a^l b^l$. Sei eine Zerlegung $w = u_1 v u_2$ mit $|v| \geq 1$, $|u_1 v| \leq l$ gegeben. Dann haben wir $v \in a^+$, und damit $u_1 v^k u_2 = u_1 u_2 \notin L$ für $k \neq 1$, da Hinzufügen von Kopien von v oder Weglassen von v die Anzahl as ändert, Die Anzahl bs aber nicht.

2. $L = \{u \in \{a, b\}^* \mid u \text{ Palindrom}\}$ ist nicht regulär, denn:

   Zu $l \geq 1$ wähle $w = a^l b a^l$. Sei eine Zerlegung $w = u_1 v u_2$ mit $|v| \geq 1$, $|u_1 v| \leq l$ gegeben. Dann ist uv ein Präfix des Präfixes a^l von w; damit ändert Hinzufügen von Kopien von v oder Weglassen von v die Anzahl as in diesem Präfix, lässt aber das Ende von w ab dem b unverändert, so dass die Palindromeigenschaft verlorengeht.

6.3 Sprachen als Kodaten

Wir beobachten, dass ein DFA A aus einem sogenannten *(deterministischen) Transitionssystem* $A_0 = (Q, \Sigma, \delta, F)$ und einem Zustand $s \in A_0$ besteht. Wir verwechseln per Currying δ mit

$$\delta : Q \to (\Sigma \to Q)$$

und fassen F als eine Abbildung $F : Q \to 2 = \{\bot, \top\}$ auf. Somit hat A_0 die Form

$$\langle F, \delta \rangle : Q \to 2 \times (\Sigma \to Q),$$

d.h. Transitionssysteme sind G-Koalgebren für

$$GX = 2 \times (\Sigma \to X).$$

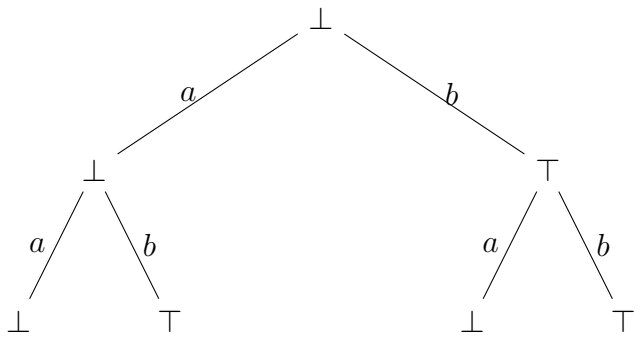
Die entsprechende Kodatentypdeklaration

```
1 codata Lang Σ where
2         acc: Lang Σ -> Bool
3         der: Lang Σ -> (Σ -> Lang Σ)
```

definiert die finale G-Koalgebra, mit wie folgt beschriebenen Elementen:

- Die Elemente sind unendliche Bäume;
- die Knoten sind in *Bool* gelabelt;
- jeder Knoten hat je ein Kind für jedes $a \in \Sigma$

Für $\Sigma = \{a, b\}$ sehen die ersten drei Ebenen so eines Baums also z.B. wie folgt aus:



Wir können die Knoten eines derartigen Baums mittels *Adressen* in Σ^* beschreiben. Der zweite Knoten dritten Ebene oben hat beispielsweise die Adresse ab. Der Baum ist dann vollständig beschrieben durch die Menge

$$\{u \in \Sigma^* \mid \text{Knoten } u \text{ hat Label } \top\};$$

d.h. ein Baum ist einfach eine Sprache. Die finale G-Koalgebrastruktur $\langle F, \delta \rangle$ übersetzt sich auf Sprachen wie folgt:

- Nach der Beschreibung der finalen Koalgebrastruktur auf unendlichen Bäumen gilt $F(L) = \top$ genau dann, wenn die Wurzel des entsprechenden Baums den Label $\top$ hat; da die Wurzel durch das leere Wort ϵ adressiert wird, ist dies genau dann der Fall, wenn $\epsilon \in L$
- Nach der Beschreibung der finalen Koalgebrastruktur auf unendlichen Bäumen ist $\delta(L)(a)$ das a-te Kind der Wurzel des Baums L. Die Adresse u in diesem Teilbaum adressiert den Knoten au im ursprünglichen Baum, d.h. wir haben

  $$\delta(L)(a) = \{u \in \Sigma^* \mid au \in L\}.$$

  Somit fängt δ eine bekannte Konstruktion auf Sprachen ein, *Ableitung* einer Sprache L nach a, die man auch $\delta_a(L)$ oder einfach L_a schreibt.

Wir bezeichnen im folgenden mit Λ sowohl die Menge aller Sprachen als auch die finale G-Koalgebra.

Nachdem wir Transitionssysteme mit G-Koalgebren identifiziert haben, erhalten wir einen natürlichen Begriff von *Morphismen* von Transitionssystemen, eben in Gestalt von Morphismen von G-Koalgebren. Für Transitionssysteme $A = (Q_A, \Sigma, \delta_A, F_A)$ und $B = (Q_B, \Sigma, \delta_B, F_B)$ ist eine Abbildung $f : Q_A \to Q_B$ demnach ein Morphismus $A \to B$, wenn das Diagramm

$$\begin{array}{ccc} Q_A & \xrightarrow{\quad f \quad} & Q_B \\ {\scriptstyle \langle F_A, \delta_A \rangle}\downarrow & \# & \downarrow{\scriptstyle \langle F_B, \delta_B \rangle} \\ 2 \times (\Sigma \to Q_A) & \xrightarrow[2 \times (\Sigma \to f)]{} & 2 \times (\Sigma \to Q_B) \end{array}$$

kommutiert (man erinnere sich, dass nach unserer Konvention 2 auch die Identität auf 2 bezeichnet). Da das Ergebnis beider Rechenpfade (in der rechten unteren Ecke) ein Paar ist, läuft dies auf zwei Gleichungen hinaus:

1. Kommutieren in der linken Komponente:

   $$F_B\Big(f(x)\Big) = F_A(x)$$

   d.h. x ist genau dann final, wenn $f(x)$ final ist; d.h. f *lässt Finalität invariant.*

2. Man erinnere sich, dass $(\Sigma \to f)(h) = f \circ h$, d.h. Kommutieren in der rechten Komponente läuft auf die Bedingung

   $$\delta_B(f(q))(a) = ((f \circ \delta_A)(q))(a) = f(\delta_A(q)(a))$$

   hinaus, d.h. f ist *verträglich mit Transitionen.* (In der üblichen Notation für DFAs schreibt sich diese Bedingung als $\delta_B(a, f(q)) = f(\delta_A(a, q))$.)

Mit Blick auf korekursive Definitionen von Sprachen beschreiben wir als Nächstes die (eindeutigen) Morphismen in die finale G-Koalgebra Λ:

Satz 6.12. *Zu $A_0 = (Q, \Sigma, \delta_A, F_A)$ ist*

$$\begin{aligned} f : Q &\to \Lambda \\ q &\mapsto L(A_0, q) \end{aligned}$$

ein Transitionssystemmorphismus. Dabei bezeichnen wir mit (A_0, q) den Automaten, der aus A_0 durch Hinzunahme von q als initialen Zustand entsteht.

In Worten: Der eindeutige Morphismus $A_0 \to \Lambda$ bildet jeden Zustand auf seine akzeptierte Sprache ab.

Zur Verwendung im Beweis des Satzes verallgemeinern wir (wie üblich) die Zustandsübergangsfunktion δ auf Worte $u \in \Sigma^*$ per

$$\begin{aligned} \delta(\epsilon, q) &= q \\ \delta(au, q) &= \delta(a, \delta(u, q)). \end{aligned}$$

Ein DFA $(Q, \Sigma, \delta, s, F)$ akzeptiert dann ein Wort u genau dann, wenn $\delta(u, s) \in F$. Für $L \in \Lambda$ schreiben wir $L_u = \delta(u, L)$.

Beweis (Satz 6.12). *f lässt Finalität invariant*: Ein Zustand $q \in Q$ ist offenbar genau dann final, wenn $\epsilon \in L(A_0, q)$, was nach der Beschreibung der Struktur von Λ wiederum gerade bedeutet, dass $L(A_0, q) = f(q)$ final ist.

f ist verträglich mit Transitionen, d.h. $f(\delta_A(a, q)) = \delta(a, f(q))$: Wir haben

$$\begin{aligned} u \in f(\delta_A(a, q)) &= L(A_0, \delta_A(a, q)) \\ &\iff \delta(u, \delta_A(a, q)) = \delta(au, q) \in F \\ &\iff au \in L(A_0, q) = f(q) \\ &\iff u \in \delta(a, f(q)). \end{aligned}$$

□

Korollar 6.13. *1. $L \in \Lambda$ akzeptiert L*

2. Morphismen von Transitionssystemen bewahren die akzeptierte Sprache; d.h. wenn $f : A_0 \to B_0$ ein Morphismus von Transitionssystemen ist, gilt $L(A_0, q) = L(B_0, f(q))$.

Beweis. Nach Satz 6.12 ist die von einem Zustand akzeptierte Sprache sein Verhalten im Sinne von Bemerkung 4.38. □

6.4 Minimierung

Wir entwickeln nun eine koalgebraische Sicht auf die Minimierung von deterministischen Automaten, d.h. die Überführung eines gegebenen Automaten einen äquivalenten Automaten (d.h. einen, der dieselbe Sprache akzeptiert) mit minimaler Anzahl Zustände.

Definition 6.14. Für einen Zustand q in einem Transitionssystem A definieren wir das von q *erzeugte Untertransitionssystem* $\langle q\rangle$ durch

$$\langle q\rangle = \{\delta(u,q) \mid u \in \Sigma^*\}.$$

Es ist klar, dass $\langle q\rangle$ abgeschlossen unter δ ist; somit ist $\langle q\rangle$ mit der von A geerbten Transitionsabbildung tatsächlich ein Transitionssystem.

Lemma 6.15. *Es gilt* $L(\langle q\rangle, q) = L(A, q)$.

Beweis. Die Einbettung $\langle q\rangle \hookrightarrow A$ ist ein Morphismus, so dass die Behauptung aus Korollar 6.13 folgt. □

Lemma 6.16. *Jeder Morphismus f von Transitionssystemen schränkt ein zu einer Surjektion*

$$f : \langle q\rangle \twoheadrightarrow \langle f(q)\rangle.$$

Insbesondere gilt $|\langle f(q)\rangle| \leq \langle q\rangle$.

(Dabei bezeichnet $|A|$ die Anzahl Zustände von A.)

Beweis. Es gilt $f(\delta(u,q)) = \delta(u, f(q))$. □

Insbesondere gilt also für jedes Transitionssystem A_0 und jeden Zustand q in A_0

$$|\langle L(A_0, q)\rangle| \leq |\langle q\rangle| \leq |A_0|$$

Wenn also eine Sprache von einem Automaten (A, q) erzeugt wird, also $L(A, q) = L$, dann ist $|A| \geq |\langle L\rangle|$, das heisst, $\langle L\rangle$ ist der *minimale Automat* für L. Dies zeigt:

Satz 6.17. *Eine Sprache L ist genau dann regulär, wenn $\langle L\rangle$ endlich ist.*

Wir erinnern nunmehr an einige mengentheoretische Konzepte. Gegeben eine Äquivalenzrelation R auf einer Menge X ist die *Äquivalenzklasse* $[x]_R$ von $x \in X$ definiert als $[x]_R = \{y \in X \mid xRy\}$. Die *Quotientenmenge* X/R ist dann gegeben als $X/R = \{[x]_R \mid x \in X\}$. Wir sagen, dass R *endlichen Index* hat, wenn X/R endlich ist. Wenn nun $f : X \to Y$ eine Abbildung ist, definiert f auf X eine Äquivalenzrelation $\operatorname{Ker} f$ durch $x(\operatorname{Ker} f)y \iff f(x) = f(y)$. Wir haben dann eine bijektive Abbildung

$$h : X/\operatorname{Ker} f \to f[X],$$

definiert durch $h([x]_R) = f(x)$. Insbesondere hat also das Bild $f[X]$ von f dieselbe Kardinalität wie $X/\operatorname{Ker} f$. Ende der Erinnerung.

Nun ist $\langle L \rangle = \{L_u \mid u \in \Sigma^*\}$ gerade das Bild der Abbildung $d : \Sigma^* \to \Lambda$ mit $d(u) = L_u$, ist also in Bijektion mit $\Sigma^*/\operatorname{Ker} d$. Wir schreiben $\sim_L := \operatorname{Ker} d$; dann gilt für $v, w \in \Sigma^*$

$$\begin{aligned} v \sim_L w &\iff L_v = L_w \\ &\iff \forall u \in \Sigma^*. (u \in L_v \Leftrightarrow u \in L_w) \\ &\iff \forall u \in \Sigma^*. (vu \in L \Leftrightarrow wu \in L). \end{aligned}$$

Damit ist also Satz 6.17 gerade der klassische Satz von Myhill und Nerode, den wir noch einmal ausdrücklich formulieren:

Satz 6.18 (Myhill und Nerode)**.** *Eine Sprache L ist genau dann regulär, wenn die durch*

$$v \sim_L w \iff \forall u \in \Sigma^*. (vu \in L \Leftrightarrow wu \in L)$$

definierte Äquivalenzrelation $\sim_L$ endlichen Index hat.

6.5 Reguläre Ausdrücke per Korekursion

Nach den Resultaten der vorigen Abschnitte können wir die Semantik regulärer Ausdrücke alternativ definieren, indem wir ein Transitionssystem $\mathcal{E} \to 2 \times (\Sigma \to \mathcal{E})$ auf der Menge $\mathcal{E}$ der regulären Ausdrücke angeben. Letzteres definieren wir wiederum *rekursiv*. Wir erinnern an die Definition regulärer Ausdrücke in Form einer Datentypdeklaration:

```
1 data Expr Σ where
2         0,1: () -> Expr Σ
3         (_): Σ -> Expr Σ
4         +, ·: Expr Σ -> Expr Σ -> Expr Σ
5         _*: Expr Σ -> Expr Σ
```

Wir definieren dann $\langle F, \delta \rangle : \mathcal{E} \to 2 \times (\Sigma \to \mathcal{E})$ rekursiv wie folgt:

- $F(0) = \bot$ und
  $\delta(0)(a) = 0$
- $F(1) = \top$ und
  $\delta(1)(a) = 0$
- $F(a) = \bot$ und
  $\delta(a)(a) = 1$, $\delta(a, b) = 0$ für $b \neq a$
- $F(r + s) = F(r) \vee F(s)$ und
  $\delta(r + s)(a) = \delta(r)(a) + \delta(s, a)$
- $F(rs) = F(r) \wedge F(s)$ und
  $\delta(rs)(a) = \begin{cases} \delta(r)(a)s & \text{falls } F(r) = \bot \\ \delta(s)(a) + \delta(r)(a)s & \text{sonst} \end{cases}$
- $F(r^*) = \top$ und
  $\delta(r^*)(a) = \delta(r)(a)r^*$

Das definiert eine Abbildung $L : \mathcal{E} \to \Lambda$ korekursiv per

- $F(L(r)) \Leftrightarrow F(r)$ (also $\epsilon \in L(r) \Leftrightarrow F(r)$)
- $\delta(L(r))(a)(= L(r)_a) = L(\delta(r)(a))$

Unter Verwendung dieser korekursiven Definition können wir nunmehr koinduktive Beweise der Äquivalenz regulärer Ausdrücke führen. Der Begriff der Bisimulation instanziiert auf das finale Transitionssystem Λ wie folgt:

Eine Relation $R \subseteq \Lambda \times \Lambda$ ist eine *Bisimulation*, wenn für alle $L\,R\,K$ gilt:

- $\epsilon \in L \iff \epsilon \in K$
- $L_a\,R\,K_a$ für alle $a \in \Sigma$

Beispiel 6.19 (Gleichheit (der erzeugten Sprachen) von regulären Ausdrücken)**.** Wir identifizieren im folgenden zur Erleichterung der Notation reguläre Ausdrücke r mit den durch sie definierten Sprachen $L(r)$, und schreiben insbesondere r_a für $L(r)_a$ u.ä.

1. Wir zeigen $r + 0 = r$ koinduktiv, indem wir zeigen, dass

   $$R = \{(r + 0, r) \mid r \in \mathcal{E}\}$$

   eine Bisimulation ist:

   - $F(r + 0) = F(r) \vee F(0) = F(r) \vee \bot = F(r)$
   - $(r + 0)_a = r_a + 0_a = r_a + 0\,R\,r_a$

2. Wir zeigen $r(s + t) = rs + rt$ koinduktiv.

   Wir versuchen zunächst zu zeigen, dass $R = \{(r(s + t), rs + rt) \mid r, s, t \in \mathcal{E}\}$ eine Bisimulation ist: Wir haben, im etwas komplizierteren Fall $F(r) = \top$,

   $$(r\ (s + t))_a = (s + t)_a + r_a\ (s + t) = s_a + t_a + r_a\ (s + t)$$

   und

   $$(rs + rt)_a = rs_a + rt_a = s_a + r_a s + t_a + r_a t = s_a + t_a + r_a s + r_a t,$$

   so dass die beiden Seiten nur bis auf Kongruenz bezüglich $+$ in Relation stehen. Wir wählen daher stattdessen allgemeiner

   $$R = \{(p + r(s + t), p + rs + rt) \mid r, s, t \in \mathcal{E}\}\,;$$

   man beachte die Analogie zur Strategie der Verstärkung von Induktionsbehauptungen. R ist nun tatsächlich eine Bisimulation:

   - Wir haben

     $$F(p + r(s + t)) = F(p) \vee (F(r) \wedge (F(s) \vee F(t)))$$

     und

     $$F(p + rs + rt) = F(p) \vee (F(r) \wedge F(s)) \vee (F(r) \wedge F(t)),$$

     was nach dem Assoziativgesetz der Aussagenlogik äquivalent ist.

- Wir schränken uns wieder auf den komplizierteren Fall $F(r) = \top$ ein, und haben dann

$$\begin{aligned}(p + r(s + t))_a &= p_a + s_a + t_a + r_a(s + t)\\ &R\ p_a + s_a + t_a + r_a s + r_a t\\ &= (p + rs + rt)_a.\end{aligned}$$