Theorie der Programmierung

31. Oktober 2014 - 05:53 Uhr

Dieses Skript zur Vorlesung *„Theorie der Programmierung“* im Sommersemester 2014 wurde von den untenstehenden Studenten erarbeitet. Es ist somit offensichtlich inoffiziell und erhebt weder einen Anspruch auf Korrektheit noch auf Vollständigkeit.

Florian Jung `florian.jung@fau.de`

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

Dieses Skript wird in einem nur angemeldeten Nutzern zugänglichen SVN gepflegt. Aus Transparenzgründen gibt es einen – noch inoffizielleren – *git-svn-Mirror*, der täglich geupdated wird:
`git clone git://git.informatik.uni-erlangen.de/thprog-skript`
https://git.informatik.uni-erlangen.de/?p=thprog-skript
Wichtig: In diesem Mirror findet keinerlei Entwicklung statt, er ist read-only.

Die jeweils aktuelle Version des Skripts ist als PDF verfügbar im WWWCIP unter https://wwwcip.cs.fau.de/~ki08jofa/#thprog.

Patches welcome, bitte an `thprog-patches@windfisch.org`.

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/Nipkour: 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:

- Hopecroft/Ullmann/Motwani

(Ko-)Induktion

- Jacobs/Rutten: A Tutorial on (Co-)Algebras and (Co-)Induction

1.2 Übungsbetrieb

Es gibt je ein Arbeitsblatt pro Kapitel. Je Blatt hat man 3 Wochen Zeit es zu lösen. Es gibt einen maximalen Bonus von 15% auf die bereits bestandene Klausur gutgeschrieben.

1.3 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

Definition 2.1. Termersetzung wird verstanden als eine sukzessive (und erschöpfende) Umformung von Termen gemäß *gerichteter* Gleichungen.

Anwendungen:

- (algebraische) Spezifikation
- Programmverifikation
- automatisches Beweisen
- Programmierung
  - Turing-vollständig
  - funktionale Programmiersprachen
- Computeralgebra (Gröbnerbasen / Buchbergeralgorithmus)

Beispiel 2.2 (Addition mit Haskell)**.**

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

Beispiel 2.3 (Umformung am Beispiel von $2+1$)**.**
`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))` ❑

Beispiel 2.4 (Distributivgesetz)**.**
Wir wollen zeigen: $(2+x)+y = 2+(x+y)$
Beweis: `plus(Suc (Suc x)) y` $\rightarrow$ `Suc(plus(Suc x) y)` $\rightarrow$ `Suc(Suc (plus x y))` ❑

Optimierung:

plus(plus x y) z = plus x (plus y z)

Beim linken Teil der Gleichung wird der x Teil zweimal verschoben. Beim rechten Teil nur einmal.

Formatierung des lstlisting unten an das texttt oben anpassen

```
1 plus x y = plus y x
2 plus (Suc x) y = plus x (Suc y)
3 Suc (plus x y)
```

Spezifikation und Verifikation:

```
1 plus(suc (Suc Zero)) x = plus (Suc Zero) (Suc x)
2 Suc(plus(Suc Zero) x)
3 Suc(Suc(plus Zero x))
```

2.1 Syntax und operationale Semantik

2.1.1 Recall: Binäre Relationen

Definition 2.5. Eine *binäre Relation* zwischen zwei Mengen X und Y ist ein $R \subseteq X \times Y$. Man schreibt $x \,\mathrm{R}\, y$, wenn $(x, y) \in R$.

Beispiel 2.6.
Die „Kleiner-Gleich"-Relation auf den natürlichen Zahlen „$\leq$" $\subseteq \mathbb{N} \times \mathbb{N}$ ist wie folgt definiert: „$\leq$" $= \{(n, m) \mid n \leq m\}$
In diesem Fall ist also $X = Y = \mathbb{N}$.

Definition 2.7. Eine Relation $R \subseteq X \times X$ heißt:

- *reflexiv*, wenn für alle $x \in X$ gilt: $x \,\mathrm{R}\, x$, d.h. $(x, x) \in R$.
- *symmetrisch*, wenn für alle $x, y \in X$ gilt: $x \,\mathrm{R}\, y \Rightarrow y \,\mathrm{R}\, x$, d.h. wenn x zu y in Relation steht, dann auch y zu x.
- *transitiv*, wenn für alle $x, y, z \in X$ gilt: Aus $x \,\mathrm{R}\, y$ und $y \,\mathrm{R}\, z$ folgt: $x \,\mathrm{R}\, z$.
- *Präordnung*, wenn R reflexiv und transitiv ist.
- *Äquivalenzordnung*, wenn R reflexiv, transitiv und symmetrisch ist.

Definition 2.8 (Beispiele für Relationen)**.**

- Gleichheitsrelation („="): $id = \{(x, x) | x \in X\} = \Delta$
- Verkettung zweier Relationen R und S: $R \circ S := \{(x, z) \mid \exists y \text{ mit } (x \,\mathrm{S}\, y \wedge y \,\mathrm{R}\, z)\}$
  Wir definieren induktiv die n-fache Verkettung einer Relation mit sich selbst:
  $R^0 := id$, $R^n := R \circ R^{n-1}$.
- $R^- = \{(x, y) \mid y \,\mathrm{R}\, x\}$. (Beispiel: „$\geq$" = „$\leq^-$")

Lemma 2.9. *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 2.10. 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.

- Reflexiver Abschluss: $R \cup id$
- Symmetrischer Abschluss: $R \cup R^-$
- Transitiver Abschluss: $R^+ := R \cup (R \circ R) \cup (R \circ R \circ R) \cup \ldots = \bigcup_{n=1}^{\infty} R^n = \{(x,y) \mid \exists n \geq 1 \text{ mit } (x,y) \in R^n\}$
  also ist $x\, R^+\, y \Leftrightarrow \exists x_0, \ldots, x_n$ mit $x = x_0 \,\mathrm{R}\, x_1 \,\mathrm{R} \ldots \mathrm{R}\, x_{n-1} \,\mathrm{R}\, x_n = y$
- Transitiv-Reflexiver Abschluss: $R^* = \bigcup_{n=0}^{\infty} R^n = R^+ \cup id = (R \cup id)^+$

Lemma 2.11 (Erzeugte Äquivalenz)**.** *$(R \cup R^-)^*$ ist eine symmetrische Relation. Allgemein gilt: Wenn S symmetrisch ist, dann sind auch S^+ und S^* symmetrisch.*

Beweis. 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. □

Beispiel 2.12.
$R \subseteq \mathbb{N} \times \mathbb{N}$ ist im folgenden die Relation, die eine natürliche Zahl mit ihrer unmittelbaren Vorgängerzahl in Beziehung setzt: $R := \{(n+1, n) \mid n \in \mathbb{N}\}$.

Die Relation $\tilde{R} := (R \cup R^-)$ ist diejenige mit $n \,\mathrm{R}\, m \Leftrightarrow |n - m| = 1$.

Dann ist $R^+ =$„$>_{\mathbb{N}}$“ und $R^* =$„$\geq_{\mathbb{N}}$“.

2.1.2 Recall: Terme

Definition 2.13.

- Eine *Signatur* Σ ist eine Menge von Funktionssymbolen $f, g, \ldots$ zusammen mit ihren Stelligkeiten.
  Schreibe: $f/n \in \Sigma$, lies: „f ist n-stellig in Σ“.
- Sei V eine Menge von Variablensymbolen. Ein *Term* ist dann induktiv definiert wie folgt:
  - Sei $x \in V$. Dann ist x, also eine einzelne Variable, ein Term.
  - Eine Konstante c, d.h. ein nullstelliges Funktionssymbol, ist 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.

- $T_\Sigma(V)$ ist die Menge aller Terme.

Definition 2.14 (freie Variablen)**.** Die Menge der freien Variablen von t: $FV(t)$ ist induktiv definiert wie folgt:

- Sei $x \in V$, dann ist x auch ein Term. $FV(x) := \{x\}$
  Das bedeutet: Die Menge der freien Variablen eines Termes, der nur eine einzelne Variable ist, enthält nur ebendiese Variable.

- Seien $t_1, \ldots, t_n$ Terme und $f/n \in \Sigma$. $FV(f(t_1, \ldots, t_n)) := \bigcup_{i=1}^{n} FV(t_i)$

Definition 2.15 (Substitution)**.** Eine *Substitutionsabbildung* ist ein $\sigma : V \to T_\Sigma(V)$, das einzelne Variablen zu (ggf. komplexeren) Termen umformt.

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

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

- $f(t_1, \ldots, t_n)\sigma := f(t_1\sigma, \ldots, t_n\sigma)$

Definition 2.16 (Kontext)**.**

1. Ein *Kontext* ist ein Term $C(\cdot)$ mit einem „Loch“ „$\cdot$“. Mit dem „Loch“ $(\cdot)$ als Komponente sind die Kontexte formal folgendermaßen induktiv definiert:

   - $(\cdot)$ ist ein Kontext.

   - Seien $t_1, \ldots, t_n \in T_\Sigma(V)$, $f/n \in \Sigma$ und sei $C(\cdot)$ ein Kontext.
     Dann ist auch $f(t_1, \ldots, t_{i-1}, C(\cdot), t_{i+1}, \ldots, t_n)$ ein Kontext.

2. Die *Auswertung* eines Kontexts $C(\cdot)$ ist rekursiv folgendermaßen definiert:
   $$C(t) = \begin{cases} t & \text{falls } C(\cdot) = (\cdot) \\ f(t_1, \ldots, \tilde{C}(t), \ldots, t_n) & \text{falls } C(\cdot) = f(t_1, \ldots, t_{i-1}, \tilde{C}(\cdot), t_{i+1}, \ldots, t_n) \end{cases}$$

Definition 2.17. 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 $\forall$ Kontexte $C(\cdot)$ und Terme s, t gilt: $t \,\mathrm{R}\, s \Rightarrow C(t) \,\mathrm{R}\, C(s)$

   - *stabil*, wenn $\forall \sigma, t, s$ gilt: aus $t \,\mathrm{R}\, s$ folgt auch $(t\sigma) \,\mathrm{R} (s\sigma)$

3. Die *Einschrittreduktion* „$\to$“ $\subseteq T_\Sigma(V) \times T_\Sigma(V)$ ist definiert als der kontextabgeschlossene stabile Abschluss von „$\to_0$“:
„$\to$“ $:= \{(C(t\sigma), C(s\sigma)) \mid t \to_0 s, C(\cdot)$ ist Kontext und σ ist Substitution.$\}$.

4. Seien s, t zwei Terme.

   - Wir sprechen von *Reduktion*, wenn $s \to t$ gilt.
   - Wir sprechen von *Konvertierbarkeit*, wenn $s \leftrightarrow t$, also $s \to t \wedge t \to s$ gilt.
     „$\leftrightarrow$“=„$\leftarrow$“$\cup$„$\leftarrow^{-}$“

5. Ein Term t heißt *normal*, wenn t zu nichts umgeformt werden kann. Schreibweise: $t \not\to$
   formal: $\nexists s$, sodass $t \to s$.

6. Ein Term t heißt eine *Normalform* eines Termes s, wenn t normal ist und es gilt: $s \to^* t$.

Beispiel 2.18.
Sei folgende Umformung erlaubt: $x + (y + z) \to_0 (x + y) + z$,
d.h. $(x + (y + z), (x + y) + z) \in \to_0$.

Betrachte nun $(a + (b + (c + d)) + e, (a + b) + (c + d) + e)$. Dies ist nicht in $\to_0$, aber in $\to$, denn:

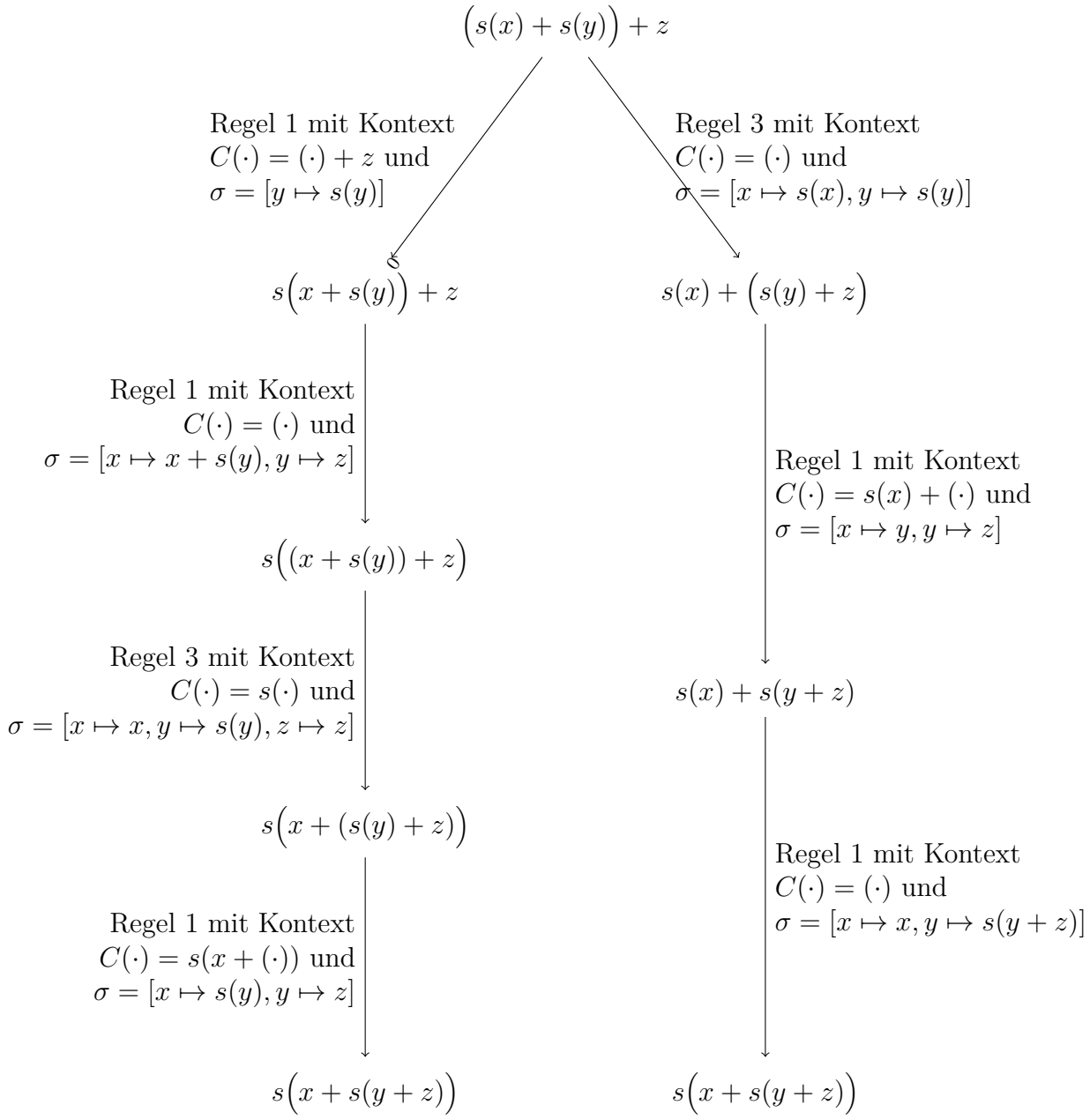
Setze $C(\cdot) := (\cdot) + e$ und $\sigma : \begin{cases} x \mapsto & a \\ y \mapsto & b \\ z \mapsto & c + d \end{cases}$.

Beispiel 2.19.
Sei $\Sigma = \{plus/2, s/1, 0/0\}$ und $\to_0$ enthalte folgende Regeln:

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

Hierbei ist $a + b$ zu verstehen als $plus(a, b)$, und die letzte Regel darf gesehen werden als $plus\Big(plus(x, y), z\Big) \to_0 plus\Big(x, plus(y, z)\Big)$.

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



Die beiden unteren Blätter des Baumes sind Normalformen. Wir sehen, dass in diesem Fall es keine Rolle spielt, welche Wahl wir am Anfang treffen; die Normalformen sind beide Male dieselben.

Definition 2.20. Eine Relation $R \subseteq X \times X$ heißt *wohlfundiert* (well-founded, w.f.) genau dann wenn es *keine* unendliche Folge $(x_i)_{i \in \mathbb{N}}$ gibt mit $x_0 \,\mathrm{R}\, x_1 \,\mathrm{R}\, x_2 \ldots$.

Beispiel 2.21.
$(\mathbb{N}, >)$ ist wohlfundiert, $(\mathbb{Z}, >)$ aber nicht.

2.1.3 Terminierung

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

Definition 2.22. Ein Term t heißt

- *schwach normalisierend* $\Leftrightarrow$ t hat Normalform $\Leftrightarrow$ $\exists s$ mit $t \to^* s$ und s normal
  D.h. Es existieren $t_1, \ldots, t_n$, mit $t \to t_1 \to \cdots \to t_n \to s$ und s normal.
- *stark normalisierend* $\Leftrightarrow$ es existiert keine unendliche Folge $(t_i)_{i\in\mathbb{N}}$ mit $t = t_0 \to t_1 \to t_2 \to \ldots$

Bemerkung 2.23.
schwach/stark normalisierend wird auch WN/SN (weakly/strongly normalizing) abgekürzt.

Definition 2.24. Ein Termersetzungssystem $(\Sigma, \to_0)$ heißt *schwach/stark normalisierend* genau dann, wenn jeder Term schwach/stark normalisierend in $\to$ ist.

Beispiel 2.25.
Es sind gegeben:

$$f(x) \to_0 f(x) \tag{4}$$
$$g(x) \to_0 1 \tag{5}$$

$g(x)$ 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, aber nicht stark normalisierend:
$g(f(3)) \underset{\sigma=[x\mapsto f(3)]}{\overset{(2)}{\longrightarrow}} 1 \not\to$
Aber $g(f(3)) \overset{(1)}{\to} g(f(3)) \to \ldots$

2.1.4 Terminierungsbeweise

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.

   (Erinnerung: kontextabgeschlossen $\Leftrightarrow$ $tRs \to C(t)\,\mathrm{R}\,C(t)$;
   es reicht schon: $C(\cdot) = f(x_1, \ldots, (\cdot), \ldots, x_n)$)

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 $\forall t, s : t \to s \Rightarrow t > s$ (also $\to$ wohlfundiert), da „>“ kontextabgeschlossen und stabil.

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

„>“ ist kontextabgeschlossen: $(|t| > |s| \Rightarrow |C(t)| > |C(s)|)$
Aber nicht stabil: $(x + 2y) - x > y + y$, aber $(x + 2t) - x \not> t + t$ für t groß.

Beispiel 2.29.
Definiere „>“ durch $t > s :\Leftrightarrow s$ ist echter Unterterm von t.

„>“ ist wohlfundiert und stabil, aber nicht kontextabgeschlossen:
$x + x > x$, aber nicht $f(x + x) > f(x)$

Beispiel 2.30.
$\emptyset$ ist Reduktionsordnung.

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

2.2 Polynomordnungen

2.2.1 Recall

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. $X^2Y + 2Y^2ZX$

Produkt und Summen von Polynomen sind Polynome, also für $p \in \mathbb{N}[X_1, \ldots, X_n]$ und $q_1, \ldots, q_n \in \mathbb{N}[Y_1, \ldots, Y_m]$ gilt:
$p(q_1, \ldots, q_n) \in \mathbb{N}[Y_1, \ldots, Y_m]$.

Definition 2.32. Definiere „$>_A$“ für nichtleeres $A \subseteq \mathbb{N}$ und für Polynome $p, q \in \mathbb{N}[X_1, \ldots, X_n]$ 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: $X^2 >_{\mathbb{N}_{\geq 1}} X$

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 $p_0 \underbrace{(a, \ldots, a)}_{\in\mathbb{N}} > p_1(a, \ldots, a) > \ldots$, was in $\mathbb{N}$ aber unmöglich ist! Widerspruch!

□

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

Lemma 2.35. *p streng monoton* $\Rightarrow \forall k_1, \ldots, k_n, l_1, \ldots, l_n : \forall j : k_j \geq l_j \wedge \exists j : k_j > l_j \Rightarrow p(k_1, \ldots, k_n) > p(l_1, \ldots, l_n)$

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

- für jedes $f/n \in \Sigma$ ein $p_f \in \mathbb{N}[X_1, \ldots, X_n]$
- $A \subseteq \mathbb{N}$, sodass aus $a_1, \ldots, a_n \in A$ folgt: $p_f(a_1, \ldots, a_n) \in A$

Die so induzierte *Polynomordnung* ist $t \succ_{\mathcal{A}} s :\Leftrightarrow p_t >_A p_s$ mit $p_x = X$. $p_{f(t_1,\ldots,t_n)} = p_f(p_{t_1}, \ldots, p_{t_n})$.

Gehört das noch zur Definition? Verstehe den $p_x := X$-Teil nicht.

Satz 2.37. *Polynomordnungen sind Reduktionsordnungen.*

Beweis.

- $\succ$ wohlfundiert, da $>_A$ wohlfundiert.

- $\succ$ stabil: klar
- $\succ$ kontextabgeschlossen: per p_f streng monoton. q.e.d.

□

Korollar 2.38. *Wenn es also für ein Termersetzungssystem T eine Polynomordnung gibt, so dass für jede Umformungsregel $t \to_0 s$ von T gilt: $t \succ_{\mathcal{A}} s$ ($\Leftrightarrow P_t >_A s$), dann ist T* stark normalisierend*, d.h. es „terminiert“[1].*

Beweis. folgt unmittelbar mit Satz 2.37 und Satz 2.27. □

Beispiel 2.39.

$$x \cdot (y + z) \to_0 x \cdot y + x \cdot z \tag{6}$$

$$(x + y) + z \to_0 x + (y + z) \tag{7}$$

$P_+ := 2x + y + 1$, $P_\cdot := xy$,
$A := \mathbb{N} \setminus \{0, 1\}$

Zu (6):
$P_{x(y+z)} = P_\cdot(x, P_+(y, z)) = X(2Y + Y + 1) = 2XY + XZ + X := (*)$
$P_{xy+xz} = P_+(P_\cdot(x, y), P_\cdot(x, z)) = 2XY + XZ + 1 <_A (*)$.

Zu (7):
$P_{(x+y)+z} = \cdots = 2(2x + y + 1) + z + 1 = 4x + 2y + z + 3 := (\#)$
$P_{x+(y+z)} = \cdots = 2x + 2y + z + 1 + 1 = 2x + 2y + z + 2 <_A (\#)$

[1] Hinweis: „terminieren“ wurde nicht exakt definiert und stellt lediglich eine Vorstellungshilfe dar.

2.3 Konfluenz

Definition 2.40. Sei T ein Termersetzungssystem.

1. Terme s, s' heißen *zusammenführbar* (z.f.), wenn ein q existiert mit $s \to^* q$ und $s' \overset{*}{\to} q$.

2. Das Termersetzungssystem T heißt *konfluent* (CR, nach Church-Rosser) genau dann, wenn für alle Terme t, s, s' mit $t \overset{*}{\to} s$ und $t \overset{*}{\to} s'$ gilt: s und s' sind zusammenführbar.

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

Bemerkung 2.41.
Konfluenz ist eine wichtige Eigenschaft von deterministischen Programmiersprachen, da sie den 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.

Beispiel 2.42.
Einige Anwendungen der Church-Rosser-Konfluenz sind:

- Programmieren: Church-Rosser-Konfluenz ≡ Determinismus.
- Computer-Algebra-Systeme: Church-Rosser-Konfluenz und und „stark normalisierend“ ⇒ Entscheidbarkeit

Satz 2.43. *Sei T ein konfluentes Termersetzungssystem T.*

1. *Für Terme t, s gilt*

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

2. *für alle Terme t, s, s' gilt: Wenn s, s' Normalformen von t sind, dann ist $s = s'$. Das „=“ bedeutet hier „syntaktisch gleich“.* („Eindeutigkeit der Normalform“)

Beweis.

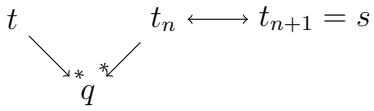
ad 1.) „⇐“ ist klar. Beweis für „⇒“:

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:

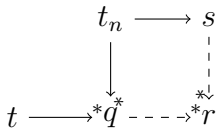
- $n = 0$: Dann gilt $s = t$, wir können also $q = t$ wählen und haben dann $t \overset{*}{\to} q \overset{*}{\leftarrow} s$, d.h. t und s sind zusammenführbar.

- $n \to n+1$: 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:



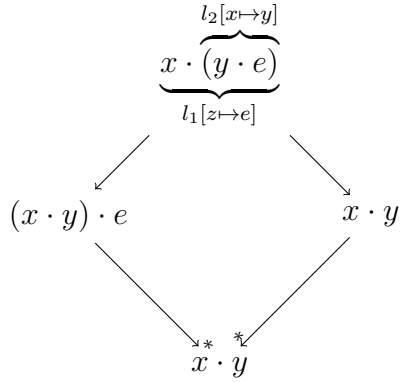
ad 2.) Dann $s \leftrightarrow^* s'$, also gibt es nach (1.) ein q mit $s \xrightarrow{*} q \leftarrow^* s'$., also $s = q = s'$, da s, s' normal sind.

□

Beispiel 2.44 (Gruppen)**.**
Die algebraische Struktur der Gruppe gibt die folgenden Umformungsregeln vor:

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

Damit kann der Ausdruck $x \cdot (y \cdot e)$ nun zu zwei unterschiedlichen Ausdrücken umgeformt werden. Allerdings kann man zeigen, dass jede der möglichen Umformungsrichtungen am Ende in derselben Normalform resultiert:



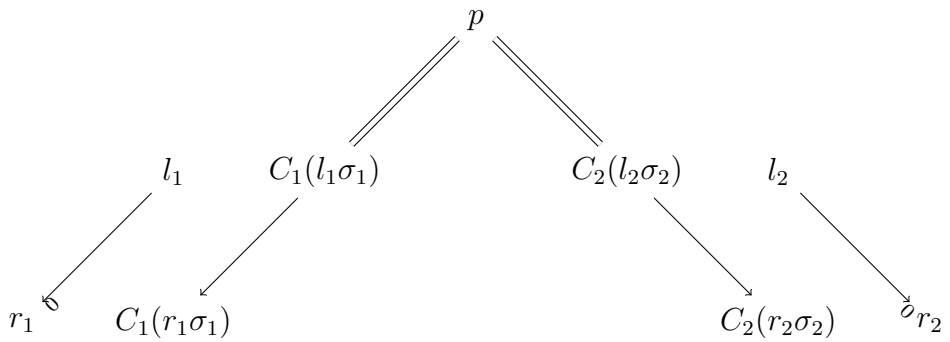
Das Termersetzungssystem ist also stark normalisierend.

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

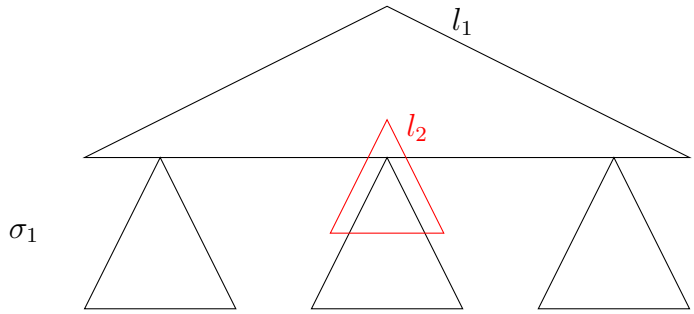
Beweis. folgt auf Seite 26. □

Im Folgenden stellen wir uns die Frage, wann ein Termersetzungssystem *lokal konfluent* (WCR) ist. Hierzu bedienen wir uns des Konzeptes der *kritischen Paare* (nach Knuth/-Bendix).

Betrachten wir den folgenden Fall, in dem ein Term p mit zwei verschiedenen Regeln reduziert werden kann:



Derartig konkurrierende Reduktionsmöglichkeiten werden insbesondere dann zum Problem, wenn die zweite Regel $l_2 \to_0 r_2$ „in“ einem Teil der ersten Regel $l_1 \to_0 r_1$ steckt, und somit durch ihre Anwendung die Anwendbarkeit von Regel 1 vernichtet 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.



Definition 2.46. 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

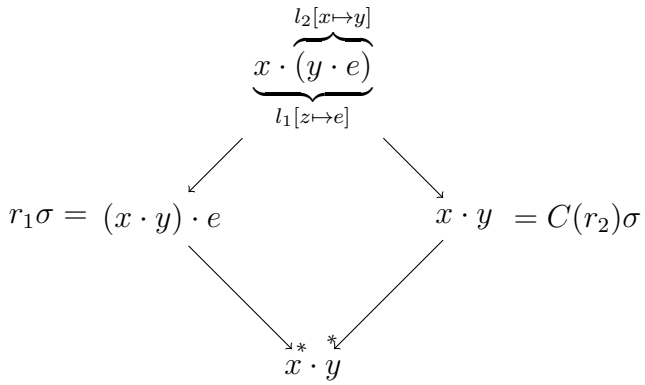
t ein nichttrivialer Term ist (d.h. $t \notin V$, t ist nicht nur eine Variable), sodass t und l_2 unifizierbar sind. Sei $\sigma = mgu(t, l_2)$. [2]

Dann heißt $(r_1\sigma, C(r_2)\sigma)$ ein *kritisches Paar*.

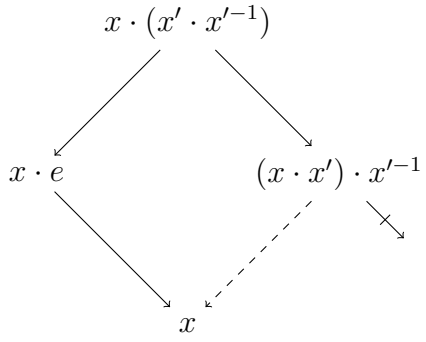
Beispiel 2.47 (Gruppen, siehe oben)**.**

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

$t = y \cdot z$ mit dem Kontext $C(\cdot) := x \cdot (\cdot)$
$\sigma = [y \mapsto x', z \mapsto e]$



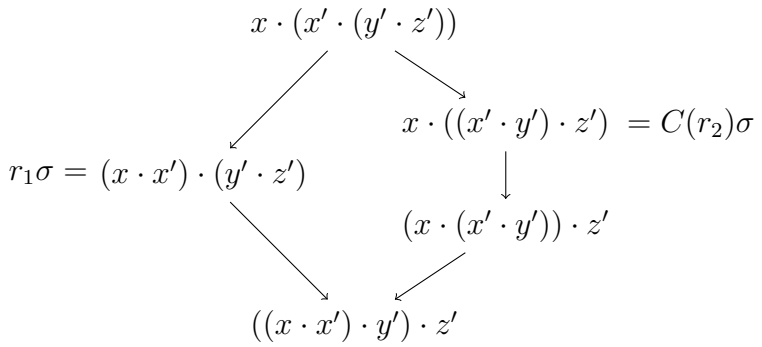
Beispiel 2.48.
mit $\sigma = mgu(y \cdot z, x' \cdot x'^{-1}) = [y \mapsto x', z \mapsto x'^{-1}]$ haben wir:



[2] most general unifier. http://en.wikipedia.org/wiki/Unification_(computer_science)

Hier liegt also keine Konfluenz vor, da der gestrichelte Pfeil durch keine Umformungsregel erlaubt wird. Um ein konfluentes System zu erhalten, müssen wir also noch eine weitere Regel hinzufügen.

Beispiel 2.49.
betrachte $x \cdot (y \cdot z)$ vs. $x' \cdot (y' \cdot z')$:
mit $\sigma = mgu\Big(y \cdot z, x' \cdot (y' \cdot z')\Big) = [y \mapsto x', z \mapsto y' \cdot z']$



Bemerkung 2.50.
Es gibt (für $\to_0$ endlich) nur endlich viele kritische Paare.

Satz 2.51 (Critical Pair Lemma)**.** *Das Termersetzungssystem T ist* lokal konfluent (WCR) *genau dann, wenn alle kritischen Paare zusammenführbar sind.*

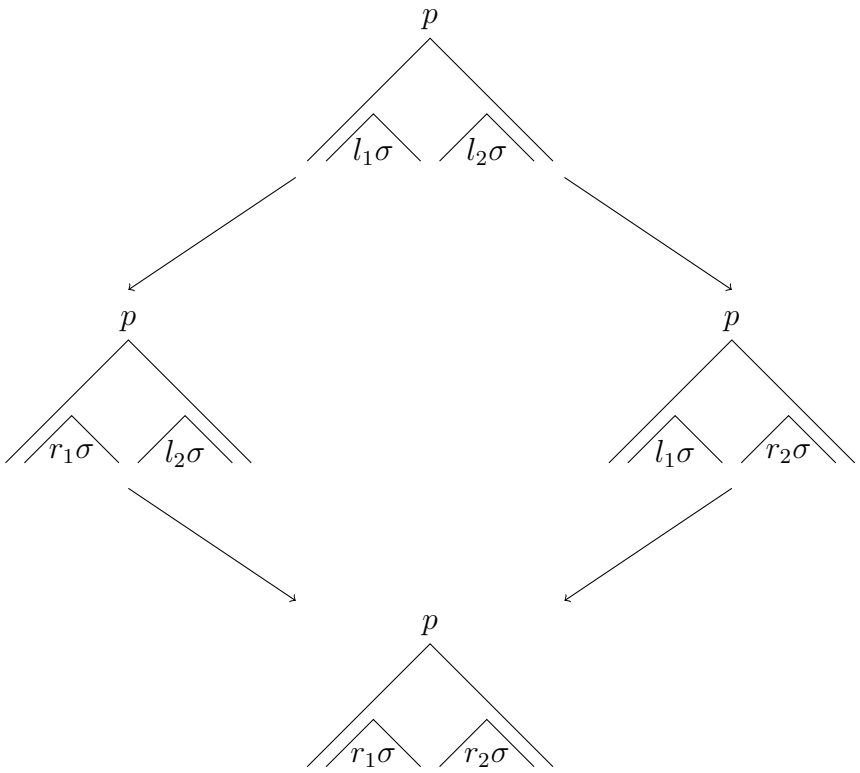
Beweis.

Definition 2.52.

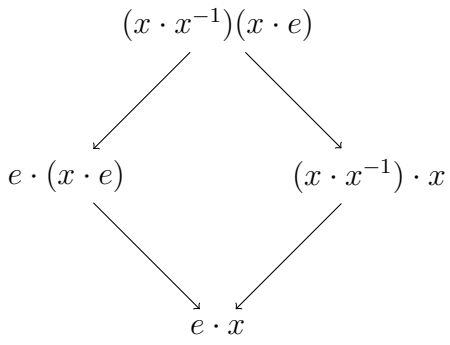
- Schreibe $C(\cdot) \sqsubseteq D(\cdot)$ wenn $\exists E(\cdot)$ mit $C(\cdot) = D(E(\cdot))$
  (d.h. „D hängt im Syntaxbaum unterhalb von C“).
- Schreibe $C(\cdot) \perp D(\cdot)$ wenn $C(\cdot) \not\sqsubseteq D(\cdot)$ und $D(\cdot) \not\sqsubseteq C(\cdot)$.

Beweis per Fallunterscheidung. Angenommen, die beiden Regeln $l_1 \to r_1$ und $l_2 \to r_2$ konkurrieren miteinander (d.h. sie können beide umgesetzt werden – eine Reihenfolge ist nicht zwingend):

1. $C_1(\cdot) \perp C_2(\cdot)$: Es ist egal, ob man zuerst die Regel $l_1 \to r_1$ (linker Pfad im Bild) oder die Regel $l_2 \to r_2$ (rechter Pfad) anwendet, da sich die beiden nicht beeinflussen:

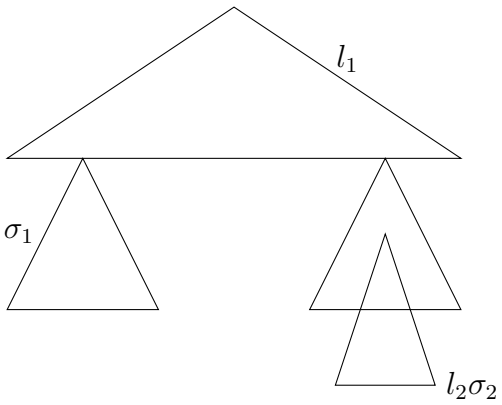


Beispiel 2.53.
Ein Beispiel hierfür ist folgende Situation im bekannten Termersetzungssystem „mathematische Gruppe“:



2. o.B.d.A: $C_2(\cdot) \sqsubseteq C_1(\cdot)$. o.B.d.A. kann $C_1(\cdot) = (\cdot)$ angenommen werden.

   a) $C_2(l_2\sigma_2) = l_1\sigma'$



o.B.d.A. können wir $\sigma_2 = [\ ] = id$ annehmen.

Beispiel 2.54.
Beispielsweise nehmen wir an, dass für den Term x in l_1 durch ein σ ein Kontext des l_2 substituiert wird (x trete in l_1 hier drei mal auf).

Dann können wir entweder (links) zuerst $l_1 \to r_1$ anwenden (x trete in r_1 hier zwei mal auf). Oder aber wir wenden zuerst ein mal $l_2 \to r_2$ an (im Bild rot markiert, rechts). Das nimmt uns jedoch (temporär) die Möglichkeit, die $l_1 \to r_1$-Regel anzuwenden, da nun nicht mehr drei identische Blöcke unter dem oberen Dreieck hängen. Weitere Anwendung der $l_2 \to r_2$-Regel jedoch schafft wieder Gleichheit und damit Zusammenführbarkeit.

Siehe Abbildung 1.

□

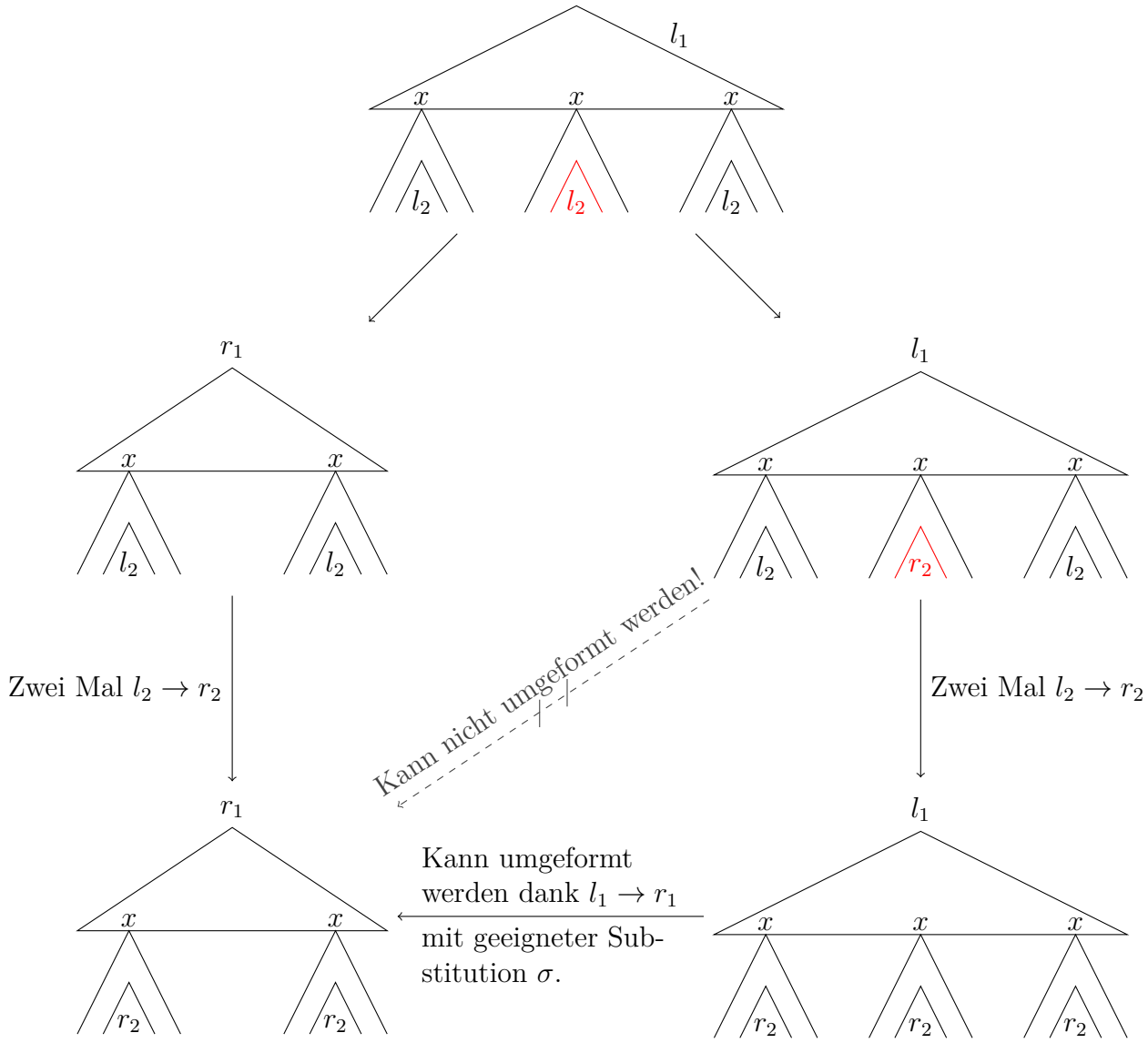


Abbildung 1: Ableitungsmöglichkeiten in Beispiel 2.54

2.4 Wohlfundierte Induktion

Satz 2.55. *Sei $R \subseteq X \times X$ wohlfundierte Relation auf einer Menge X. Dann gilt folgendes Prinzip der* wohlfundierten Induktion*:*

(*) Falls gilt: $\forall x.(\underbrace{\forall y.x \,\mathrm{R}\, y \Rightarrow P(y)}_{\textit{Induktionsvoraussetzung}}) \Rightarrow P(x)$ $\quad \leftarrow$ *Induktions-Schritt.*

(**) Dann folgt: $\forall x.P(x)$

Beweis. Angenommen, (**) gelte nicht, also $\exists x_0$ mit $\neg P(x_0)$. Dann (wegen (*)) $\exists x_1$ mit $x_0 \,\mathrm{R}\, x_1$ mit $\neg P(x_1)$.

Iterieren gibt $x_0 \,\mathrm{R}\, x_1 \,\mathrm{R}\, x_2 \,\mathrm{R} \ldots$, was ein Widerspruch zur Wohlfundiertheit von R ist. □

Beispiel 2.56.
Über die wohlfundierte Induktion lassen sich auch die anderen Induktionsarten konstruieren:

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

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

2. $R \subseteq T_\Sigma(V) \times T_\Sigma(V)$:

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

   R ist wohlfundiert, das gibt die *strukturelle Induktion*:
   Lemma 2.57. *R wohlfundiert $\Rightarrow$ R^+ wohlfundiert.*

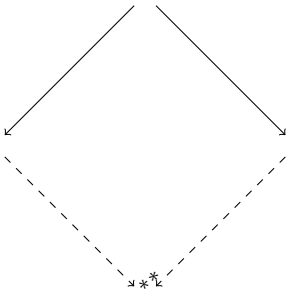
   Beweis. Angenommen, es gäbe eine unendliche R^+-Kette $\underbrace{x_0 \,R^+\, x_1}_{x_0 \overset{R}{\to} \tilde{x}_1 \overset{R}{\to} \tilde{x}_2 \overset{R}{\to} \ldots \overset{R}{\to} x_1}$ $R^+ \, x_2 \, R^+ \ldots$.
   Dann gäbe es auch eine unendliche R-Kette, Widerspruch. □

3. „$>$" ist wohlfundiert auf $\mathbb{N}$. Das gibt die *course-of-values-Induktion*:

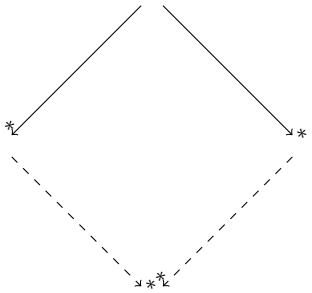
   $$\forall n : \forall m : (n > m \Rightarrow P(m)) \Rightarrow \forall n : P(n).$$

Beweis von Newmans Lemma (Satz 2.45). (stark normalisierend und lokal konfluent $\Rightarrow$ CR-konfluent)

Erinnerung:



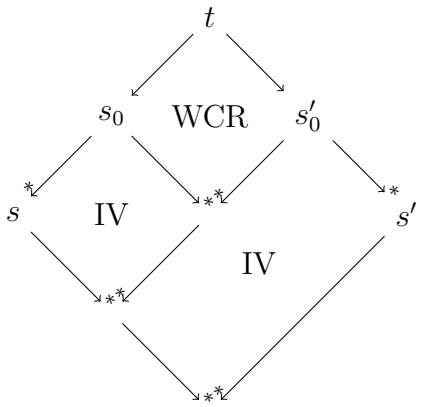
lokale Konfluenz



Church-Rosser-Konfluenz

Sei $t \xrightarrow{*} s, s'$, zu zeigen: s, s' sind zusammenführbar.

$o.B.d.A : s \neq t \neq s'$



wohlfundierte Induktion über t bezüglich $\rightarrow$. □

2.4.1 Recall: Unifikation

Definition 2.58. Eine Substition σ heißt *Unifikator* von Termen t, s, wenn $t\sigma = s\sigma$.

$Unif(t, s) := \{\sigma \mid \sigma \text{ ist Unifikator von } t, s\}$

t, s heißen *unifizierbar* genau dann, wenn $Unif(t, s) \neq \emptyset$.

Beispiel 2.59.

1. $g(x, h(x))$ und $g(h(y), y)$ sind nicht unifizierbar: Denn ein $\sigma \in Unif$ muss erfüllen: sowohl $x \mapsto h(y)$ als auch $y \mapsto h(x)$. Widerspruch!
2. $f(g(x, y), z)$ und $f(v, h(w))$ sind unifizierbar mit $\sigma := [v \mapsto g(x, y), z \mapsto h(w)]$

Definition 2.60. $\sigma = mgu(t,s) \Leftrightarrow \forall \tau : (\tau \in Unif(t,s) \Leftrightarrow \exists \tau' \text{ mit } \tau = \sigma\tau')$, d.h. σ ist „allgemeiner“ als alle anderen $\tau \in Unif(t,s)$.

Beispiel 2.61 (Wie zeigt man Konfluenz eines Termersetzungsystems?)**.**
Grundlegende Strategie: Critical-Pair-Lemma: lokale Konfluenz gilt genau dann, wenn alle kritischen Paare zusammenführbar sind. Dann: dank starker Normalisierbarkeit (ist auch zu zeigen!) folgt aus der lokalen Konfluenz sogar die Konfluenz.

Wie findet man kritische Paare? Kritische Paare treten auf, wo sich zwei Grundreduktionen überlappen. (siehe buntes Bild, TODO).

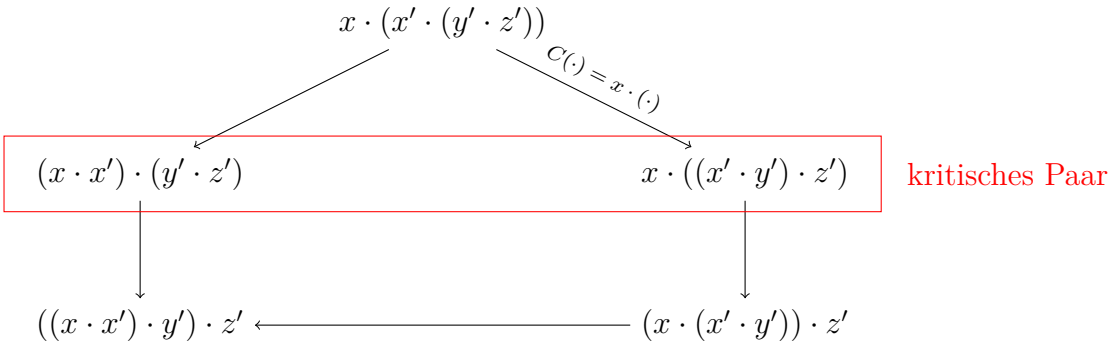
buntes Bild malen

$l_1 := x \cdot (y \cdot z)$
$l_2 := x' \cdot (y' \cdot z')$ (Achtung: sieht „gleich“ aus, aber die Variablennamen müssen disjunkt sein).

Suche jetzt einen Teilterm, der unifiziert: $y \cdot z$.

$mgu(y \cdot z, x' \cdot (y' \cdot z')) = [y \mapsto x', z \mapsto y' \cdot z']$ (wir haben ein kritisches Paar gefunden).

Hierauf haben wir den Unifikator angewandt:



3 Der λ-Kalkül (Church/Kleene)

3.1 Laudatio

Der λ-Kalkül ist eine der größte Erfindungen der Menschheit, auf gleicher Stufe wie die Quantenphysik, Relativitätstheorie, der Ottomotor oder die Currywurst.

Es gibt Dinge, die bedeutsam sind, unabhängig davon, ob sie es eines Tages als Sprachfeature in Java 8 schaffen.

3.2 Was ist der λ-Kalkül?

- 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 ist Lisp.
- $\equiv$ anonyme Funktionen und sonst nichts

3.3 Der ungetypte λ-Kalkül

Grundsatz: „Alles ist eine Funktion“. Applikation (= Funktionsaufruf) kann „von allem auf alles“ geschehen.

Der binärer Operator _ _ („_“ ist hier Platzhalter und „ “ ist der Operator) wird für diese Applikation benutzt:

- fx bedeutet „Wert von f bei Eingabe x“
- $(fx)y$ bedeutet „Wert von f bei Eingabe (x, y) (Currying)

Bemerkung 3.1 (Erinnerung: Currying)**.**
Currying erlaubt es, eine Funktion mit der Signatur $f : A \times B \to C$ zu schreiben als $(\text{curry}\, f) : A \to (B \to C)$, wobei $(\text{curry}\, f)(a) : B \to C$ mit $b \mapsto f(a, b)$.

Definition 3.2 (λ-Abstraktion)**.** Sei t ein Term.

$\lambda x.t$ bedeutet „die Funktion, die x auf t abbildet“. (Hierbei ist typischerweise $x \in FV(t)$).

Beispiel 3.3.

$\lambda x.3 + x$	ist „die Funktion, die auf ihre Eingabe 3 addiert“. (Hinweis: mit unserem derzeitigen Wissen ist „+“ noch nicht bekannt, die „Definition“ damit nicht wohlgeformt.)
$\lambda x.xx$	„verdoppelt“ ihre Eingabe.
$\lambda x.\lambda y.x$	bildet die Eingabe x ab auf diejenige konstante Funktion mit der Wertemenge $\{x\}$.

Definition 3.4 (Terme)**.** Ein *Term* ist induktiv definiert durch folgende Regeln:

- eine Variable x ist ein Term.
- die Aneinanderreihung von Termen $t_1 t_2$ ist ein Term.
- der λ-Ausdruck $\lambda x.t$, wobei x Variable und t Term ist, ist ein Term.

Die Grammatik der gültigen λ-Terme ist also: $t ::= x | t_1 t_2 | \lambda x.t$, wobei t_1 und t_2 gültige λ-Terme sind.

Definition 3.5 (Freie und gebundene Variablen)**.** Sei t ein Term.

1. Dann ist die *Menge der freien Variablen* $FV(t)$ in t rekursiv definiert wie folgt:
   - $FV(x) := \{x\}$ (wobei x Variable ist.)
   - $FV(ts) := FV(t) \cup FV(s)$
   - $FV(\lambda \underset{\substack{\uparrow \\ \text{gebunden}}}{x}.t) := FV(t) \setminus \{x\}$
2. Eine Variable x heißt *frei* in einem Term t, wenn $x \in FV(t)$.
3. Eine Variable x heißt *gebunden* in einem Term t, wenn sie nicht frei ist.

Definition 3.6 (Substitution)**.** Eine Substitution ist eine Abbildung $\sigma : V \to T(V)$, die Variablen auf Terme abbildet. Die Anwendung einer Substitution σ auf Ausdrücke ist rekursiv definiert wie folgt:

- $x\sigma = \sigma(x)$
- $(ts)\sigma = (t\sigma)(s\sigma)$
- $(\lambda x.t)\sigma = \lambda x.(t\sigma)$ wenn x Fixpunkt von σ ist ($\sigma(x) = x$) und $x \notin FV(\sigma|_{FV(t)})$

Bemerkung 3.7.

⚠ $(\lambda x.y)[y \mapsto x] \neq \lambda x.x$

Bemerkung 3.8.

Im Folgenden werden diese Konventionen benutzt:

- Auswertungen erfolgen linksassoziativ: $xyz = (xy)z$
- Kurzform für mehrere λs: $\lambda xy.t = \lambda x.\lambda y.t$
- Der Scope eines λ ist immer so groß wie möglich: $\lambda x.xx = \lambda x.(xx) \neq (\lambda x.x)x$

Folgende Äquivalenzen dürfen zur Umformung von λ-Ausdrücken verwendet werden:

3.3.1 α-Äquivalenz

Definition 3.9. Zwei Terme t_1, t_2 heißen *α-äquivalent*, wenn sie durch Umbenennung gebundener Variablen auseinander hervorgehen. Man schreibt dann $t_1 =_\alpha t_2$.

Formal: $\lambda x.t =_\alpha \lambda y.t[x \mapsto y]$, *wenn* $y \notin FV(t)$.

($\overset{!}{\triangle}$ z.B. $\lambda x.y \neq_\alpha \lambda y.y$).

Beispiel 3.10.
Mittels α-äquivalenter Umformung lässt sich das verbotene „Einfangen“ von Variablen bei Substitutionen umschiffen: $(\lambda x.y)[y \mapsto x] = (\lambda x'.y)[y \mapsto x] = \lambda x'.x$

Beispiel 3.11.
$\lambda x.xy = \lambda z.zy$

3.3.2 β-Reduktion

Die β-Reduktion entspricht dem „Ausrechnen eines Funktionsaufrufs“:

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

Das λ-Kalkül ist im Wesentlichen ein Termersetzungssystem (modulo α-Äquivalenz) mit folgenden Grundreduktionen:

(Grundreduktion für β)	$(\lambda x.t)x \to_0 t$
insbesondere	$\underbrace{(\lambda x.t)s}_{\beta\text{-Redex}} \to_\beta t[x \mapsto s]$
(manchmal auch (η)	$\lambda x.yx \to_\eta y$ („Compileroptimierung“))

Beispiel 3.12.

- $(\lambda x.xx)(yx) \to_\beta (yx)(yx) = yx(yx)$

- Nichtterminierung: Setze $\Omega := \lambda x.xx$. Dann ist $\Omega\Omega = (\lambda x.xx)\Omega \to_\beta \Omega\Omega \to_\beta \ldots$
- Booleans: true $:= \lambda xy.x$, false $:= \lambda xy.y$
- Paare:

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

Dann ist z.B. $fst(\text{pair}\ xy) = \text{fst}\left((\lambda xy.\lambda z.zxy)xy\right)$
$\to_\beta \text{fst}\left((\lambda y.\lambda z.zxy)y\right)$
$\to_\beta \text{fst}\left(\lambda z.zxy\right)$
$= \left(\lambda p.p\ \text{ true}\right)(\lambda z.zxy)$
$\to_\beta (\lambda z.zxy)\ \text{ true}$
$\to_\beta \text{true}\ xy = (\lambda xy.x)xy$
$\to_\beta \ldots \to_\beta x$

3.3.3 Rekursion

Rekursion kann im λ-Kalkül erreicht werden durch das Definieren von Fixpunktgleichungen:

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

mit $Ff = \lambda n.\ \texttt{if n=0 then}\ 1\ \texttt{else}\ n \cdot (f(n-1))$

F ist hier ein *Funktional*. $F : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$

$\text{fact} = \lambda n.\ \texttt{if n=0 then 1 else}\ n \cdot \text{fact}(n-1) =: F(\text{fact})$
$\text{fact} = \text{fix}\ F$

allgemein: $F(\text{fix}\ F) = \text{fix}\ F$

Satz 3.13. *Im ungetypten λ-Kalkül:*

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

Beweis.

1. Setze $W := \lambda x.t(xx)$, $s := WW$.

   Dann $s = WW = \Big(\lambda x.t(xx)\Big)W \to_\beta t(WW) = ts$

2. Nach 1. reicht $Y = \lambda f.\Big(\lambda x.f(xx)\Big)\lambda x.f(xx)$.

□

Korollar 3.14. *Für jeden Term $s[f,x]$ existiert ein t mit $tx \to_\beta^* s[t,x]$.*

Beweis. Setze $t := Y\Big(\lambda f.\lambda x.s[f,x]\Big)$. Denn es ist:

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

Bemerkung 3.15.
$s[f,x]$ ist ein Term mit den freien Variablen f und x.

3.3.4 Auswertungsstrategien

Beispiel 3.16.
$\underbrace{(\lambda xy.x)x}_{\to_\beta \lambda y.x}(\overset{\text{terminiert nicht}}{\overset{\downarrow}{\Omega\Omega}})$ mit $\Omega := \lambda x.xx$ (denn dann ist $\Omega\Omega \to_\beta \Omega\Omega \to_\beta \dots$).

Dieser Ausdruck terminiert *abhängig von der Auswertungsstrategie*. Während man wohl intuitiv meint, dass der Ausdruck nicht terminiert (weil er ja $\Omega\Omega$ enthält), terminiert er mit der Auswertungsstrategie von Haskell sehr wohl. Denn das „Ergebnis“ von $\Omega\Omega$ wird sowieso verworfen (da es Input für eine konstante Funktion ist).

Definition 3.17. $(\lambda x.t)s \mapsto_\beta t[x \mapsto s]$

Definition 3.18. Die *applikative Reduktion* „$\to_a$“ ist induktiv definiert durch:

- $t \to_a s$, wenn $t \mapsto_\beta s$
- $\lambda x.t \to_a \lambda x.s$, wenn $t \to_a s$.
- $ts \to_a ts'$, wenn $s \to_a s'$
- $ts \to_a t's$, wenn s normal ist und $t \to_a t'$

Definition 3.19. Die *normale Reduktion* „$\to_n$“ ist definiert durch

- $t \to_a s$, wenn $t \mapsto_\beta s$ (wie oben)
- $\lambda x.t \to_a \lambda x.s$, wenn $t \to_a s$ (wie oben)
- $ts \to_n t's$, wenn $t \to_n t'$
- $ts \to_n ts'$, wenn $s \to_n s'$ und t normal und keine λ-Abstraktion ist.

Sie heißt auch „*leftmost-outermost*“-Reduktion.

Bemerkung 3.20.
Die applikative Reduktion führt im Beispiel 3.16 zu Nichttermination, während die normale (leftmost-outermost) Reduktion terminieren würde.

Definition 3.21. Eine Reduktion $t \to_\beta t' \to_\beta t'' \to_\beta \ldots \to_\beta s$ (schreibe: $t \to_\beta^* s$) heißt *erfolgreich*, wenn sie in einer Normalform s endet.

Satz 3.22 (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. *En-Gros-Reduktion* „$\to_g$“:

   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) $(\lambda x.t)s \to_g t'[x \mapsto s']$, wenn $t \to_g t'$ und $s \to_g s'$

   Lemma 3.23. *Die transitive und reflexive Hülle der en-Gros-Reduktion ist gleich die der bereits bekannten β-Reduktion: $\to_g^* = \to_\beta^*$*
   Lemma 3.24. *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* „$\to_w$“:

   allgemein: $t = t_1 \ldots t_n$ sodass t_1 entweder Variable oder λ (aber keine Applikation) ist.

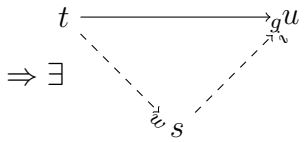
   Dann wird reduziert wie folgt: $(\lambda x.t)s_1 s_2 \ldots s_n \to_w t[x \mapsto s_1]s_2 \ldots s_n$.

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 $i \in \{1, \dots, n\}$.

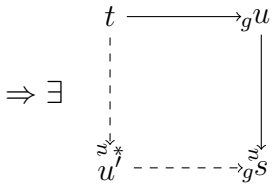
   b) t_1 Variable oder λ-Abstraktion ist.

Lemma 3.25. *Wenn es eine en-Gros-Reduktion $t \to_g u$ gibt, dann gibt es einen Term s, sodass es eine Folge von weak-head-Reduktionen $t \to_w^* s$ und s in einem Schritt intern auf u reduziert, d.h. $s \to_i u$:*



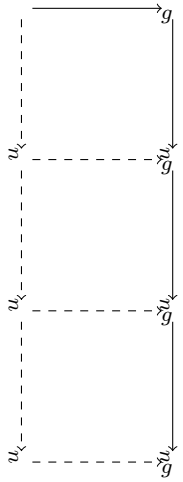
Lemma 3.26.

1.



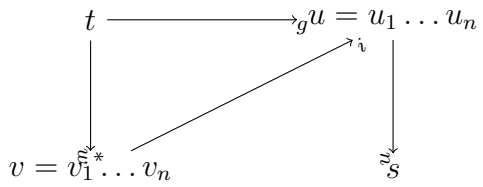
2. Dasselbe gilt, wenn man $u \to_u s$ durch $u \to_u^* s$ ersetzt.

Beweis. 2. aus 1. per Induktion:



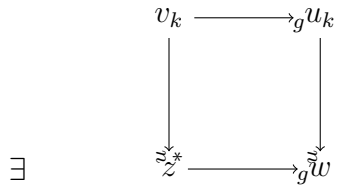
1. Induktion über u.

per Lemma 3.25:



mit v_1 (i) Variable oder (ii) λ-Abstraktion. Hier nur (i).

$u_1, \dots, u_{k-1}$ Normalformen, $s = u_1 \dots u_{k-1} w u_{k+1} \dots u_n$ mit $u_k \to_n w$. per IV:



per Lemma 3.24: $v_i \to_n^* u_i$, $i \in \{1, \dots, k-1\}$

also

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

□

Damit kann nun der Beweis der Standardisierung geführt werden:

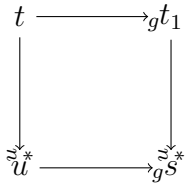
Sei $t \to_\beta^* s$, s Normalform.

per Lemma 3.23: $t = t_0 \to_g \dots \to_g t_n \to_g s$

Induktion über n:

- falls $n = 0$ ✓ per Lemma 3.24.
- falls $n > 0$: per Induktionsvoraussetzung gilt:

  per Lemma 3.25: $\exists$



  per Lemma 3.24: $u \to_n^* s$.

□

Klarstellen, dass x immer Variablen und t immer Terme sind

Bemerkung 3.27.
Warum ist es manchmal dennoch sinnvoll, die applikative Reduktion anzuwenden? Betrachte:

$$\Big(\lambda x.fxx\Big)\Big((\lambda xy.xy)ts\Big)$$

Hier verdoppelt sich der hintere Term bei normaler Reduktion anfangs, und man hat gegebenenfalls sehr große Ausdrücke mitzuführen.

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

Definition 3.28. Für den *Typ* der Funktion von α nach β schreibt man: $\alpha \to \beta$.

Beispiel 3.29.
betrachte beispielsweise $\lambda x.x : \underset{\text{Typvariable}}{\underset{\uparrow}{a}} \to a$

Definition 3.30. Sei $\mathbf{V}$ eine Menge von Typvariablen.

Die Grammatik der Typen ist dann $\alpha, \beta ::= a|\mathbf{b}|\alpha \to \beta$ wobei $\mathbf{b} \in \mathbf{B}$ Basistyp, $a \in \mathbf{V}$

Bemerkung 3.31.
Funktionen mit mehreren Argumenten können durch *Currying* (siehe Bemerkung 3.1) erreicht werden:

$$\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \beta \equiv \alpha_1 \to \Big(\alpha_2 \to \ldots \to (\alpha_n \to \beta)\Big)$$

Beispiel 3.32 (Typisierung nach Church)**.**
$\lambda x : \alpha \,.\, t$ (Wird hier nicht verwendet.)

Definition 3.33 (Typisierung nach Curry)**.** Ein *Kontext* ist eine Menge von „Variable gehört Typ an"-Beziehungen in der folgenden Art:

$\Gamma = \{x_1 : \alpha_1; \ldots; x_n : \alpha_n\}$ (die Variablen x_i sind paarweise verschieden).

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

$$\underset{FV(t)\subseteq\underset{\uparrow}{\{x_1,\ldots,x_n\}}}{\Gamma \vdash t : \alpha}$$

Es gelten die folgenden drei Regeln:

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

(Axiom: Wenn $x : \alpha \in \Gamma$, so kann daraus gefolgert werden, dass x den Typ α hat.)

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

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

Beispiel 3.34.
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.

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

Nun füllen wir diese Lücken von oben nach unten:

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

Beispiel 3.35.
Nicht jeder Term ist typisierbar! Beispielsweise:

$\lambda x.xx =: \Omega$, d.h. es gibt kein Γ und kein α, sodass gilt: $\Gamma \vdash \Omega : \alpha$, d.h. sodass im Kontext Γ Ω vom Typ α ist.

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

- 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 3.36.

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

3.4.1 Elementare Eigenschaften

Lemma 3.37 (Weakening)**.** *Wenn aus einem Kontext von Annahmen Γ ein $t : \alpha$ hergeleitet werden kann, dann auch von jedem Kontext mit zusätzlichen Annahmen $\Gamma' \supseteq \Gamma$. In Formeln:*

$\Gamma \vdash t : \alpha, \Gamma \subseteq \Gamma' \Rightarrow \Gamma' \vdash t : \alpha$

Beweis. Induktion über Herleitung von $\Gamma \vdash t : \alpha$

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

   Dann auch $x : \alpha \in \Gamma'$, also

   $$\frac{}{\Gamma' \vdash x : \alpha}$$

2. Ende der Herleitung:

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

   Nach IV: $\Gamma' \vdash t : \beta \to \alpha$, $\Gamma' \vdash s : \beta \overset{(\to_e)}{\Longrightarrow} \Gamma' \vdash ts : \alpha$.

3. Ende der Herleitung:

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

Nach IV: Da $\Gamma, x : \alpha \subseteq \Gamma', x : \alpha$ (Anmerkung: $x \notin \Gamma'$, sonst umbenennen)
$\Gamma, x : \alpha \vdash t : \beta \overset{(\to_i)}{\Rightarrow} \Gamma' \vdash \lambda x.t : \alpha \to \beta$.

□

Lemma 3.38 (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. □

3.4.2 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. Offenbar ist ersteres aber allgemeiner und daher zu bevorzugen.

Definition 3.39. σ heißt *allgemeinste Lösung* von $\Gamma \vdash t : \alpha$, wenn σ die allgemeinste Typsubstitution mit $\Gamma\sigma \vdash t : \alpha\sigma$

σ (bzw. $\sigma(a)$) heißt *Prinzipaltyp* von (Γ, t), wenn σ die allgemeinste Lösung von $\Gamma \vdash t : a$ ist, wobei a „frisch“ ist, d.h. nicht im Kontext Γ vorkommt.

Algorithmus 3.40 (Algorithm W nach Hindley/Milner)**.**
Wir bauen $PT(\Gamma; t; \alpha)$ auf als die Menge von Typgleichungen, sodass der *most general unifier* $\sigma = mgu\Big(PT(\Gamma; t; \alpha)\Big)$ die allgemeinste Lösung von $\Gamma \vdash t : \alpha$ sei, *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

mal weiter oben irgendwo „frisch“ definieren

Der *Prinzipaltyp* von (Γ, t) ist dann $mgu(PT(\Gamma; t; a))(a)$

Beispiel 3.41.
$PT(0;\ \lambda xy.xy;\ a)$ man würde als Ergebnistyp $(a \to b) \to a \to b$ erwarten.
$= PT(x : b;\ \lambda y.xy;\ c) \cup \{a \doteq b \to c\}$
$= PT(x : b, y : d;\ xy;\ e) \cup \{a \doteq b \to c;\ c \doteq d \to e\}$

$= PT(x : b, y : d;\ x;\ f \to e) \cup PT(x : b, y : d;\ y;\ f) \cup \{a \doteq b \to c;\ c \doteq d \to e\}$
$= \{b \doteq f \to e;\ d \doteq f, c \doteq d \to e, a \doteq b \to c\} =: \epsilon$

$mgu(\epsilon) = [f \mapsto d, b \mapsto d \to e, c \mapsto d \to e, a \mapsto (d \to e) \to d \to e]$

also hat $\lambda xy.xy.$ den Prinzipaltyp $mgu(\epsilon)(a) = (d \to e) \to d \to e$.

Beispiel 3.42.

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

3.4.3 Subjekt-Reduktion

Lemma 3.43 (Substitutionslemma)**.**

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

Beweis.

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

□

Satz 3.44 (Subjektreduktion)**.** $\Gamma \vdash t : \alpha, t \to^* s \implies \Gamma \vdash s : \alpha$
⚠ *die Umkehrung gilt nicht, z.B.* $(\lambda xy.y)\lambda x.xx \implies \lambda y.y$ underbrace

Beweis. ohne Einschränkungen gelte $t \to s$.

Betrachte zunächst $t \to_\beta s$, d.h. $t = (\lambda x.u)v, s = [x \mapsto v]u$

$\Gamma \vdash (\lambda x.u)v : \alpha \underset{inv}{\implies} \exists\beta$ sodass $\Gamma \vdash \lambda x.u : \beta \to \alpha$ und $\Gamma \vdash v : \beta \underset{inv}{\implies} \Gamma, x : \beta \vdash tu : \alpha$ underbrace
$\underset{Subst.lemma}{\implies} \Gamma \vdash tu[x \mapsto v] : \alpha$

Rest per Induktion über Kontexte $C(\cdot)$, z.B.
$C(t)s \to C(t')s : \Gamma tC(t)s : \alpha$

$\underset{inv}{\Longrightarrow} \exists\beta.\Gamma \vdash C(t) : \beta \to \alpha$
$\underset{IV}{\Longrightarrow} \Gamma \vdash C(t') : \beta \to \alpha \implies \Gamma \vdash C(t')s : \alpha$ □

3.4.4 Der Curry-Howard-Isomorphismus

untertitel „Programs are proofs“

„Minimale Logik“ /Implikationsfragment der intuitonistischen propositionalen Logik IPL
$\varphi, \psi ::= a | \varphi \to \psi \qquad a \in V$
„Types are propositions“

igitt. Ganze Sätze bilden.

natürliches Schließen:

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

Alternativ: Sequentenkalkül mache lokale Annahmen Γ explizit.
$\Gamma \vdash \varphi$ φ ist herleitbar aus lokaler Annahme Γ

- $\to_E \frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$
- $\to_I \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}$
- (Ax) $\frac{}{\Gamma \vdash \varphi}$, falls $\varphi \in \Gamma$

Satz 3.45. $\vdash \varphi \Leftrightarrow \varphi$ *inhabited*

Beweis. $\Rightarrow$: Streichen der Terme aus Herleitung von $\vdash t : \varphi$ gibt Herleitung von $\vdash \varphi$

$\Leftarrow$: Zu $\Gamma = \{\varphi_1, ..., \varphi_n\}$ setze $\overline{\Gamma} = \{x_1 : \varphi_1, ..., x_n : \varphi_n\}$

Zeige per Induktion über Herleitung von $\Gamma \vdash \varphi$ $\exists t.\overline{\Gamma} \vdash t : \varphi$

(Ax) $\frac{}{\Gamma \vdash \varphi}$, falls $\varphi \in \Gamma$. Dann $\varphi = \varphi_i$ für ein i, und $\frac{}{\overline{\Gamma} \vdash x_i : \varphi_i}$.

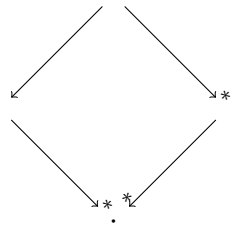
($\to_E$) $\frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$ Nach IV existieren t, s mit $\frac{\overline{\Gamma} \vdash t : \varphi \to \psi \qquad \overline{\Gamma} \vdash s : \varphi}{\overline{\Gamma} \vdash ts : \psi}$

($\to_I$) $\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}$ nach IV $\frac{\overline{\Gamma}, x_{n+1} : \varphi \vdash t : \psi}{\Gamma \vdash \lambda_{n+1}.t : \varphi \to \psi}$ □

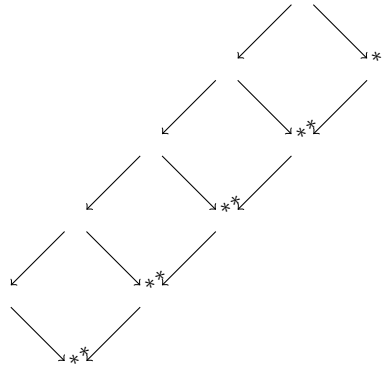
3.4.5 Church-Rosser im λ-Kalkül

Diese Section schöner machen

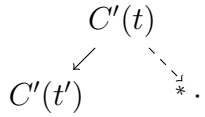
Lemma 3.46 (Streifenlemma)**.**



Das reicht für CR:



Idee:



Dazu: *markierte* Terme

$t, s := x|ts|\lambda x.t|(\lambda \underline{x}.t)s$ β-Reduktion wie für λ

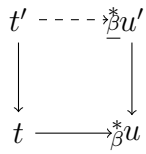
$|t| = t$ ohne Markierung $(|(\underline{\lambda} x.t) = (\lambda x.|t|)|s|)$

$$\begin{aligned} \varphi(x) &= x \\ \varphi(ts) &= \varphi(t)\varphi(s) \qquad \text{reduziere alle unterstrichenen } \beta\text{-Redexe} \\ \varphi(\lambda x.t) &= \lambda x.\varphi(t) \\ \varphi((\lambda x.t)s) &= \varphi(t)\,[x \mapsto \varphi(s)] \end{aligned}$$

Notation: $t \xrightarrow{|\cdot|} |t| \qquad t \xrightarrow{\varphi} \varphi(t)$

Beschriftung an den Pfeilen, erklärung

Lemma 3.47.

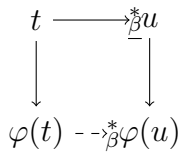


Lemma 3.48.

a) $t[x \mapsto s][y \mapsto u] = t[y \mapsto u][x \mapsto s[y \mapsto u]]$

b) $\varphi(t[x \mapsto s]) = \varphi(t)[x \mapsto \varphi(s)]$

c)



a) *Beweis.* Induktion über Struktur von t. Interessant nur $t = (\underline{\lambda} x.u)\, v$

fix alignment underset

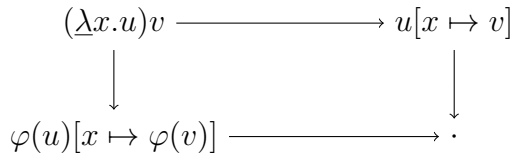
$$\begin{aligned}
\varphi((\underline{\lambda} y.u)v)[x \mapsto s]) &= \varphi((\underline{\lambda} y.u[x \mapsto s])v[x \mapsto s]) \qquad \text{ohne Einschränkung } y \notin FV(s) \\
&\underset{Def.\varphi}{=} \varphi(u[x \mapsto s])[y \mapsto \varphi(v[x \mapsto s])] \\
&\underset{2xIV}{=} \varphi(u)[x \mapsto \varphi(s)][y \mapsto \varphi(v)[x \mapsto \varphi(s)]] \\
&\underset{a)}{=} \underline{\varphi(u)[y \mapsto \varphi(v)}[x \mapsto \varphi(s)] \qquad \text{ohne Einschränkung } x \neq y, y \notin FV(\varphi(s)) \\
&\underset{Def.\varphi}{=} \varphi(\underline{(\underline{\lambda} y.u)v)}[x \mapsto \varphi(s)]
\end{aligned}$$

□

b) ohne Einschränkung $t \to s$

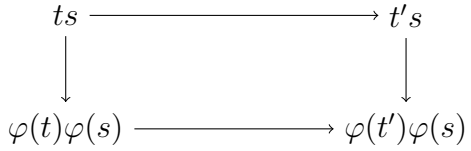
Zunächst $t \mapsto_\beta s$:

(i)

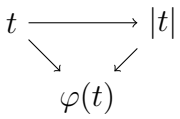


(ii)

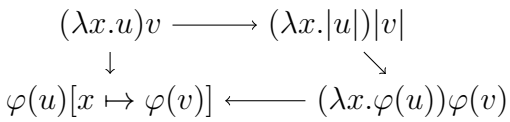
Rest per Induktion über Kontexte: z.B. label nach IV



Lemma 3.49.

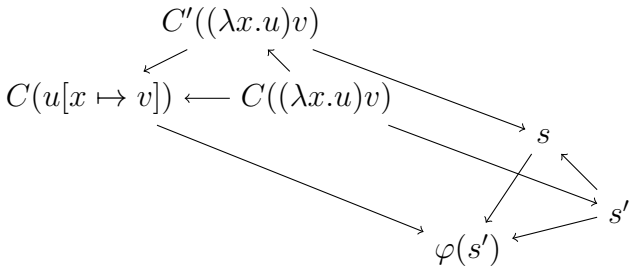


Beweis. Induktion über t, interessanter Fall:



□

Damit Beweis Streifenlemma:



Korollar 3.50. *Jeder λ-Term hat höchstens eine NF.*

3.5 Starke Normalisierung für $\lambda \to$

Via „Pseudosemantik“ [3] $[\![\ldots]\!]$ für Typen als Teilmenge von $SN := \{t \in \Lambda \mid t \text{ stark normalisierend}\}$, wobei Λ die Menge aller λ-Terme ist.

Definition 3.51. Sei $A, B \subseteq \Lambda$. Dann definieren wir $A \to B := \{t \in \Lambda \mid \forall s \in A \text{ ist } ts \in B$

Bemerkung 3.52.
Diese Schreibweise ist rein syntaktisch, aber unsemantisch.

Damit $[\![a]\!] := SN$
$[\![\alpha \to \beta]\!] := [\![\alpha]\!] \to [\![\beta]\!]$

Definition 3.53. $A \subseteq SN$ heißt *saturiert*, wenn

1. $xt_1 \ldots t_n \in A$ für alle $t_1, \ldots, t_n \in SN$ (d.h. t_i stark normalisierend), $n \geq 0$.
2. $t[x \mapsto s]u_1 \ldots u_n \in A \Rightarrow (\lambda x.t)su_1 \ldots u_n \in A$ für alle $s \in SN$

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

Lemma 3.54 (Saturiertheitslemma)**.** $[\![\alpha]\!] \in SAT$ *für alle* α.

Beweis. Induktion über α

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

   a) zu zeigen: $t_1, \ldots, t_n \in SN \Rightarrow xt_1 \ldots t_n \in SN$, d.h. wenn alle t_i starknormalisierend sind, dann auch die Anwendung von x auf die t_i.

   alle Reduktionen finden innerhalb der t_i statt, d.h. es können keine neuen β-Redexe entstehen, die möglicherweise zu Endlosrekursion führen könnten. ✓.

   b) Sei $s \in SN$ und $t[x \mapsto s]u_1 \ldots u_n \in SN \Rightarrow t \in S$.

   zu zeigen: $v := (\lambda x.t)su_1 \ldots u_n \in SN$

   Da $t, s, u_1, \ldots, u_n \in SN$ (d.h. alle beteiligten Terme sind stark normalisierend), muss jede unendliche Reduktionssequenz von v die Form $(\lambda x.t)su_1 \ldots u_n \xrightarrow{*} (\lambda x.t')s'u_1' \ldots u_n' \to t'[x \mapsto s']u_1' \ldots u'n \to \ldots$.
   $\uparrow$ $t[x \mapsto s]u_1 \ldots u_n \in SN$, Widerspruch

[3] kein angemessener Ausdruck, deshalb in Anführungszeichen

2. Seien $A := [\![\alpha]\!], B := [\![\beta]\!]$ saturiert.

   a) zu zeigen: $A \to B \subseteq SN$. Sei $t \in A \to B$.

      Dann ist die einzelne Variable $x \in A$, da A saturiert ist (siehe **??**.) $\Rightarrow tx \in B \subseteq SN \Rightarrow t \in SN$. ✓ referenz

   b) Seien $r_1, \ldots, r_n \in SN$, zu zeigen: $xr_1 \ldots r_n \in A \to B$

      Sei also $s \in A$, zu zeigen ist $xr_1 \ldots r_n s \in B$. ✓, da $s \in SN$, weil es aus A stammt, und A saturiert ist, d.h. $A \subseteq SN$ und B saturiert.

   c) Sei $s \in SN$, $t[x \mapsto s]r_1 \ldots r_n \in A \to B$. Zu zeigen: $(\lambda x.t)sr_1 \ldots r_n \in A \to B$.

      Sei also $v \in A$, zu zeigen ist dann $(\lambda x.t)sr_1 \ldots r_n v \in B$. $\underset{B \text{ sat.}}{\Leftarrow} t[x \mapsto s]sr_1 \ldots r_n v \in B \Leftarrow (\lambda x.t)sr_1 \ldots r_n \in A \to B$ und $v \in A$.

□

Definition 3.55 (Erfülltheit/Konsequenz)**.** Sei σ eine Substitution (d.h. eine Interpretation einer freien Variablen). Dann:

$$\begin{aligned} \sigma \models t : \alpha &:\Leftrightarrow t\sigma \in [\![\alpha]\!] \\ \sigma \models \Gamma &:\Leftrightarrow \sigma \models x : \alpha \forall (x : \alpha) \in \Gamma \\ \Gamma \models t : \alpha &:\Leftrightarrow \forall \sigma.(\sigma \models \Gamma \Rightarrow \sigma \models (t : \alpha)) \end{aligned}$$

$\models$ bedeutet „erfüllt“.

Beispiel 3.56.

Sei das Modell $\mathfrak{M}$ dasjenige Modell 

, in dem es nur einen Punkt a gibt, der mit sich selbst durch P in Relation steht, d.h. $P(a, a) = \text{true}$ gilt in $\mathfrak{M}$. Offenbar gilt $\mathfrak{M} \models \forall x \exists y.\ P(x, y)$.

Sei $\mathfrak{N}$ nun aber das Modell 

, dann gilt offenbar *nicht* $\mathfrak{N} \models \forall x \exists y.\ P(x, y)$.

Lemma 3.57 (Korrektheit)**.** $\Gamma \vdash t : \alpha \ \Rightarrow\ \Gamma \models t : \alpha$

(Lies: „Wenn man im Kontext Γ herleiten kann, dass t den Typ α hat, dann ist $t : \alpha$ eine logische Folgerung aus Γ in der Pseudosemantik.“)

Beweis. Induktion über die Herleitung von $\Gamma \vdash t : \alpha$

$$\Gamma \vdash x : \alpha$$

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

Sei $\sigma \models \Gamma$. Nach IV ist $\sigma \models t : \alpha \to \beta$, $\sigma \models s : \alpha$, d.h. $t\sigma \in [\![\alpha]\!] \to [\![\beta]\!]$, $s\sigma \in [\![\alpha]\!]$, also $\underbrace{(t\sigma)s\sigma}_{=(ts)\sigma} \in [\![\beta]\!]$

$\Rightarrow \sigma \models ts : \beta$. ✓.

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

Sei $\sigma \models \Gamma$. Zu zeigen ist: $(\lambda x.t)\sigma \in [\![\alpha]\!] \to [\![\beta]\!]$.

Sei also $v \in [\![\alpha]\!]$, zu zeigen: $\Big((\lambda x.t)\sigma\Big)v \in [\![\beta]\!]$.

$\Leftarrow t\sigma[x \mapsto v] \in [\![\beta]\!]$
$\uparrow$
$[\![\beta]\!]$ sat.,o.B.d.A. x frisch
$\Leftrightarrow \sigma[x \mapsto v] \models t : \beta$
$\Leftarrow \sigma[x \mapsto v] \models \Gamma, x : \alpha$
$\uparrow$
IV
$\Leftrightarrow \sigma[x \mapsto v] \models x : \alpha$
$v \in [\![\alpha]\!]$ ✓.

□

Damit beweisen wir den folgenden

Satz 3.58. *$\lambda \to$ ist stark normalisierend.*

Beweis. Sei $\Gamma \vdash t : \alpha$ (wir nehmen also an, es gibt einen typisierbaren Term t).

mit Korrektheit folgt: $\Gamma \models t : \alpha$. Setze $\sigma := [\ ]$. Dann gilt $\sigma \models \Gamma$. (Denn $x \in [\![\alpha]\!] \forall \alpha$, da $[\![\alpha]\!]$ saturiert ist.)
$\Rightarrow \sigma \models t : \alpha$
$\Rightarrow t = t\sigma \in [\![\alpha]\!] \subseteq SN$. □

4 Induktive Datentypen

Beispiel 4.1.

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

Definition 4.2.

1. Beispiel 4.1 definiert eine *Signatur* Σ_{Nat}, also eine Menge von *Funktionsymbolen*. Diese werden *Konstruktoren* genannt.

2. Eine *Semantik* von Nat ist definiert als $[\![\text{Nat}]\!] = T_{\Sigma_{\text{Nat}}}(\emptyset)$.

   Im Beispiel 4.1 ist also $[\![\text{Nat}]\!] = \{0, \text{Suc}(0), \text{Suc}(\text{Suc}(0)), \dots\}$.

Definition 4.3. Ein Σ-Modell $\mathfrak{M}$ besteht aus

- *Grundbereich* (oder Träger oder Universum), d.h. Menge M
- Für jede n-stellige Funktion f, also $f/n \in \Sigma$ $\mathfrak{M}[\![f]\!] : M^n \to M$

Bemerkung 4.4.
Σ_{Nat}-Modelle sind Σ_{Nat}-Algebren:

$[\![\text{Nat}]\!]$ wird Σ_{Nat}-Algebra per
$[\![\text{Nat}]\!][\![0]\!] = 0$
$[\![\text{Nat}]\!][\![\text{Suc}]\!](t) = \text{Suc}(t)$

Definition 4.5. Ein Σ_{Nat}-*Homomorphismus* zwischen Σ_{Nat}-Algebren $\mathfrak{M}, \mathfrak{N}$
$f : \mathfrak{M} \to \mathfrak{N}$ ist eine Funktion $f : M \to N$ mit:

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

Bemerkung 4.6 (Erinnerung)**.**
Man schreibt $m \equiv n \pmod 4 \Leftrightarrow 4 \mid (m - n)$
$[n]_4 := \{m \in \mathbb{Z} \mid m \equiv n \pmod 4\}$
Das ist *wohldefiniert*: $n_1 \equiv n_2 \pmod 4 \Rightarrow n_1 + 1 \equiv n_2 + 1 \pmod 4$

$\sim$ sei *Äquivalenzrelation* auf X. Dann ist $X/\sim = \{[x]_\sim \mid x \in X\}$
wobei $[x]_\sim = \{y \mid y \sim x\}$.

Damit können wir nun das folgende Beispiel betrachten:

Beispiel 4.7.
$N = \mathbb{Z}/4\mathbb{Z} = \Big\{[0]_4, [1]_4, [2]_4, [3]_4\Big\}$

$\mathfrak{N}[\![0]\!] = [0]$
$\mathfrak{N}[\![\mathrm{Suc}]\!][n]_4 = [n{+}1]_4$, das heißt $\mathfrak{N}[\![\mathrm{Suc}]\!]$ ist diejenige Funktion, die um eins inkrementiert, modulo 4.

Mit $\mathfrak{M} := [\![\mathrm{Nat}]\!]$ wie oben können wir nun $f : \mathfrak{M} \to \mathfrak{N}$ definieren mit:

$$\begin{aligned} f &: \mathfrak{M} \to \mathfrak{N} \\ f &: M \to N \\ &\quad t \mapsto [n]_4, \text{ wobei } t = \mathrm{Suc}^n(0) \end{aligned}$$

Im folgenden prüfen wir nach, ob das ein Σ_{Nat}-Homomorphismus zwischen $\mathfrak{M} = [\![\mathrm{Nat}]\!]$ selbst und $\mathfrak{N}$ ist:

$f\Big(\mathfrak{M}[\![0]\!]\Big) = f(0) = f\Big(\mathrm{Suc}^0(0)\Big) = [0] = \mathfrak{N}[\![0]\!]$ ✓
$f\Big(\mathfrak{M}[\![\mathrm{Suc}]\!](\mathrm{Suc}^n(0))\Big) = f\Big(\mathrm{Suc}^{n+1}(0)\Big) = [n{+}1]_4 = \mathfrak{N}[\![\mathrm{Suc}]\!]([n]_4) = \mathfrak{N}[\![\mathrm{Suc}]\!]\Big(f\Big(\mathrm{Suc}^n(0)\Big)\Big)$
✓.

$\Rightarrow f$ ist ein Homomorphismus.

Definition 4.8. Eine Σ_{Nat}-Algebra heißt *initial*, wenn für jede andere Σ_{Nat}-Algebra $\mathfrak{N}$ genau ein Σ_{Nat}-Homomorphismus $[\![\mathrm{Nat}]\!] \to \mathfrak{N}$ existiert.

Satz 4.9. *$T_\Sigma(\emptyset)$ mit $\mathfrak{M}[\![f]\!](t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ ist initiale Σ-Algebra.*

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

Beweis. $f : [\![\mathrm{Nat}]\!] \to \mathfrak{N}$ ist Σ_{Nat}-Homomorphismus $\Leftrightarrow$
$f\Big(\underbrace{[\![\mathrm{Nat}]\!][\![0]\!]}_{=0}\Big) = \mathfrak{N}[\![0]\!]$ und
$f\Big(\underbrace{[\![\mathrm{Nat}]\!][\![\mathrm{Suc}]\!](t)}_{=\mathrm{Suc}(t)}\Big) = \mathfrak{N}[\![\mathrm{Suc}]\!]\Big(f(t)\Big)$
$\Leftrightarrow f$ ist Termauswertung in $\mathfrak{N} : f(t) = \mathfrak{N}[\![t]\!]$. □

Satz 4.10. *Die initiale Σ-Algebra ist eindeutig bis auf Isomorphie.*

Beweis. Seien $\mathfrak{M}, \mathfrak{N}$ initiale Σ-Algebren.

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

per Eindeutigkeit: $g \circ f = id$ und $f \circ g = id$. $\square$

Definition 4.11 (Notation zu Mengenkonstruktionen)**.** Seien X_1, X_2 Mengen. (Denke an „Typen wie in C“, wie `int` oder `bool`):

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

Sei $f_i : X_i \to Y_i$, $i \in \{1,2\}$.

$$\begin{array}{lllll} f_1 \times f_2 : & X_1 \times X_2 \to & Y_1 \times Y_2, & \Big(f_1 \times f_2\Big)(x_1, x_2) & := \Big(f_1(x_1), f_2(x_2)\Big) \\ f_1 + f_2 : & X_1 + X_2 \to & Y_1 + Y_2, & \Big(f_1 + f_2\Big)(i, x) & := \Big(i, f_i(x)\Big) \end{array}$$

$$\begin{array}{ll} \text{Sei } g_i : X_i \to Z \text{ mit } i \in \{1,2\} : & [g_1, g_2] : X_1 + X_2 \to Z, \quad [g_1, g_2](i, x) := g_i(x) \\ \text{Sei } h_i : Z \to X_i \text{ mit } i \in \{1,2\} : & \langle h_1, h_2 \rangle : Z \to X_1 \times X_2, \quad \langle h_1, h_2 \rangle(z) := \Big(h_1(z), h_2(z)\Big) \end{array}$$

$$\begin{array}{rll} in_i : & X_i \to X_1 + X_2, & in_i(x) := (i, x) \\ \Pi_i : & X_1 \times X_2 \to X_i, & \Pi_i(x_1, x_2) := x_i \\ 1 : & 1 \to 1 & id : X \to X \end{array}$$

$$(f \circ g)(x) := f\Big(g(x)\Big)$$

Beispiel 4.12.

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

Σ_{tree}-Algebra: $\alpha := [\mathfrak{M}[\![\text{Nil}]\!], \mathfrak{M}[\![\text{Node}]\!]] : 1 + M \times M \to M$

Für einen Σ_{tree}-Homomorphismus $f : \mathfrak{M} \to \mathfrak{N}$ kommutiert also das folgende Diagramm:

$$\begin{array}{ccc} 1 + M \times M & \xrightarrow{1 + f \times f} & 1 + N \times N \\ \Big\downarrow{\scriptstyle \alpha} & & \Big\downarrow{\scriptstyle \beta} \\ M & \xrightarrow[f]{} & N \end{array}$$

Hierbei sind α und β definiert durch $\mathfrak{N}[\![\text{Node}]\!] = \beta \circ in_2$ und $\mathfrak{M}[\![\text{Node}]\!] = \alpha \circ in_2$.

$f \circ \mathfrak{M}[\![\text{Node}]\!] =$
$f \circ \alpha \circ in_2 = \beta \circ (1 + f \times f) \circ in_2 = \beta \circ in_2 \circ (f \times f) = \mathfrak{N}[\![\text{Node}]\!](f \times f)$

Lemma 4.13.

- $[f_1, f_2] \circ in_i = f_i$

  Beweis. $[f_1, f_2]\Big(in_i(x)\Big) = [f_1, f_2](i, x) = f_i(x)$ □

- $f_1 + f_2 = [in_1 \circ f_1, in_2 \circ f_2]$

  Beweis. $[in_1 \circ f_1, in_2 \circ f_2](i, x) = in_i\Big(f_i(x)\Big) = (i, f_i(x)) = (f_1 + f_2)(i, x)$ □

Beispiel 4.14 (Homomorphie per Diagramm: z.B. für Node)**.**

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

f ist Homomorphismus $\Leftrightarrow \forall x, y \in M$ gilt $f(\mathfrak{M}[\![Node]\!](x, y)) = \mathfrak{N}[\![Node]\!](f(x), f(y)) \Leftrightarrow$
$f \circ \mathfrak{M}[\![Node]\!] = \mathfrak{N}[\![Node]\!] \circ (f \times f)$.

Beispiel 4.15 (Σ-Algebren als Abbildungen: z.B. Σ_{Tree})**.**

$$\begin{array}{ccc} 1 & \xrightarrow{\quad \mathfrak{M}[\![Nil]\!] \quad} & M \\ & \searrow^{\mathfrak{M}[\![Node]\!]} & \uparrow \\ & M \times M & \end{array} \qquad 1 + M \times M \xrightarrow{\quad [\mathfrak{M}[\![Nil]\!], \mathfrak{M}[\![Node]\!]] \quad} M$$

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

Initialität = Definition per fold.

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

Auf Origami Programming hinweisen

Was man da möchte ist *primitive Rekursion*.

Beispiel 4.16.
$\text{fact}\, 0 = 1$
$\text{fact}(\text{Suc}n) = (\text{Suc}n) \cdot (\text{fact}\, n)$
Zugriff direkt auf n ist mit foldn nicht möglich!

Diese Version der Fakultätsfunktion lässt sich nicht per fold schreiben.

Beispiel 4.17.
Dasselbe mit fold:

$\langle \text{fact}, id \rangle(0) = (1, 0)$
$\langle \text{fact}, id \rangle(\text{Suc}n) = \Big(\text{Suc}\Big(\text{snd}\langle \text{fact}, id \rangle(n)\Big) \cdot \text{fst}\langle \text{fact}, id \rangle(n), \text{Suc}\Big(\text{Suc}\langle \text{fact}, id \rangle(n)\Big)\Big)$

Allgemein:

Definition 4.18. Eine *primitiv rekursive Definition* besteht aus Gleichungen der Form

$f\Big(\underbrace{c(x_1, \ldots, x_n)}_{\text{Konstruktor-Pattern}}\Big) = g\Big(f(x_1), \ldots, f(x_n), x_1, \ldots, x_n\Big)$ für jeden Konstruktor $c/n \in \Sigma$.

Satz 4.19. *Das definiert f eindeutig.*

4.1 Mehrsortigkeit

Beispiel 4.20.

```
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
```

Das definiert eine *sortierte Signatur*, mit $S = \{a, \text{Tree}\ a, \text{Forest}\ a\}$, $S_0 = \{a\}$ und $F = \{\text{Leaf} : a \to \text{Tree}\, a,\ \text{Node} : \text{Forest}\, a \to \text{Tree}\, a,\ \text{Nil} : \text{Forest}\, a,\ \text{Cons} : \text{Tree}\, a \times \text{Forest}\, a \to \text{Forest}\, a\}$.

Definition 4.21. 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* mit *Profilen* $c : a_1 \times \cdots \times a_n \to b$ mit $n \geq 0$, $a_1, \ldots, a_n \in S$ und $b \in S \setminus S_0$

Definition 4.22.

- Ein *Kontext* ist eine Menge $\Gamma = \{x_1 : a_1,\ \ldots,\ x_k : a_k\}$ mit $a_i \in S$.
- *Sortierte Terme* im Kontext Γ:

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

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

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

- (Mehrsortige) Σ-Algebren:

  Mengen $\mathfrak{M}[\![a]\!]$, $a \in S$

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

- Σ-Homomorphismen $g = (g_a : \mathfrak{M} \to \mathfrak{N})_{a \in S}$, $g_a : \mathfrak{M}[\![a]\!] \to \mathfrak{N}[\![a]\!]$ mit $g_a = id$ für $a \in S_0$.

Definition 4.23 (Initiale Σ-Algebra $\mathfrak{M}$)**.** Gegeben seien Mengen V_a, $a \in S_0$.

Setze dann $\Gamma = \{x : a \mid a \in S_0, x \in V_a\}$

$\mathfrak{M}[\![a]\!] = T_\Sigma(\Gamma)_a$, $a \in S_0$
$\mathfrak{M}[\![c]\!](t_1, \ldots, t_n) = c(t_1, \ldots, t_n)$

Satz 4.24 (Initialität (=Folding))**.** *Für alle Σ-Algebren $\mathfrak{N}$ mit $\mathfrak{N}[\![a]\!] = V_a$ (mit $a \in S_0$) existiert* genau ein *Σ-Homomorphismus $g : \mathfrak{M} \to \mathfrak{N}$*

Beispiel 4.25.
für Tree a mit $a =$ Nat:

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

$g : \mathfrak{M} \to \mathfrak{N}$

4.2 Strukturelle Induktion auf Datentypen

Induktion als Beweisprinzip für rekursive Funktionen, gemäß dem Slogan: „folge der rekursiven Definition!“.

Beispiel 4.26.

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

N.B.: Das ist eine primitiv rekursive Definition.
$\text{cc} : \text{List } a \to (\text{List } a \to \text{List } a)$
$\text{cc } Nil = \lambda k.k$
$\text{cc } (\text{Cons } xl) = \lambda k.\, \text{Cons } \underline{x}\Big(\underline{(\text{cc } l)}k\Big) = g(\text{cc } l, x, l)$

$\to$ „Primitive Rekursion übers erste Argument“ (analog übers n-te Argument)

Behauptung 4.27. $\text{cc } l(\text{cc } kv) = \text{cc}(\text{cc } lk)v$

Beweis. Falsch wäre zum Beispiel die Induktion über k: $\text{cc } (\text{cc } l(\text{Cons } xk))v$

Heuristik: l ist auf beiden Seiten in „rekursiver Position“. $\overset{\text{Slogan}}{\to}$ Induktion über l

$\text{cc } Nil(\text{cc } kv) = \text{cc } kv$
$\text{cc } (\text{cc } Nilk)v = \text{cc } kv$

$\text{cc } (\text{Cons } xl)(\text{cc } kv) = \text{Cons } x(\text{cc } l(\text{cc } kv))$

$\text{cc } (\text{Cons } al)(\text{cc } kv) = \text{Cons } x(\text{cc } l(\text{cc } kv)) \overset{IV}{=} \text{Cons } x(\text{cc } (\text{cc } lk)v) =: (\#)$

$\text{cc } (\text{cc } (\text{Cons } al)k)v = \text{cc}(\text{Cons } x(\text{cc } lk))v = \text{Cons } x(\text{cc } (\text{cc } lk)v) = (\#)$ ✓. □

4.3 Induktion über mehrsortige Datentypen

```
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 eine primitiv rekursive Funktion immer *gleichzeitig* für Tree a und Forest a. Das nennt man *gegenseitige Rekursion*

betrachte nun eine Funktion „mirror“:

```
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) = cc (mirrorf f) ( Cons(mirrort t) Nil )
```

(cc ist hierbei definiert wie oben, mit beliebiger Funktion auf Tree a).

betrachte außerdem die Funktion „flatten“:

```
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) = cc (flattenf t) (flattenf f)
```

diese Funktionen lassen sich sogar als *fold* schreiben:

$(\text{flattent}, \text{flattenf}) = \text{fold}([\_], id, \text{Nil}, cc)$

Behauptung 4.28. $\text{flattent}(\text{mirrort}\, t) = \text{reverse}(\text{flattent}\, t)$

Bemerkung 4.29.
Induktion verlangt eine *zweite* Behauptung über Forest a (zusätzlich zur „trivialen Behauptung“ T über a):
$\text{flattenf}(\text{mirrorf}\, f) = \text{reverse}(\text{flattenf}\, f)$

Beweis. $\text{flattent}(\text{mirrort}(\text{Leaf}\ x)) = \text{flattent}(\text{Leaf}\ x) = \texttt{[x]} =: (*)$
$\text{reverse}(\text{flattent}(\text{Leaf}\ x)) = \text{reverse}\ \texttt{[x]} = \ldots\ \texttt{[x]} = (*)\checkmark$

$\text{flattenf}(\text{mirrorf}\ \text{Nil}) = \text{flattenf}\ \text{Nil} = \text{Nil} =: (**)$
$\text{reverse}(\text{flattenf}\ \text{Nil}) = \text{reverse}\ \text{Nil} = \text{Nil} = (**)\checkmark$

$\text{flattent}(\text{mirrort}(\text{Node}\ f)) = \text{flattent}(\text{Node}(\text{mirrorf}\, f)) = \text{flattenf}(\text{mirrorf}\, f) \overset{\text{IV für Forest}}{\overset{\downarrow}{=}}$
$\text{reverse}(\text{flattenf}\, f) =: (\#)$
$\text{reverse}(\text{flattent}(\text{Node}\, f)) = \text{reverse}(\text{flattenf}\, f) = (\#)\checkmark$

$\text{flattenf}(\text{mirrorf}(\text{Cons}\ tf)) = \text{flattenf}(\text{cc}(\text{mirrorf}\, f)\,\texttt{[mirrort t]})$
$\overset{\text{Lemma 4.30}}{\overset{\downarrow}{=}} \text{cc}(\text{flattenf}(\text{mirrorf}\, f))(\text{flattenf}\ \texttt{[mirrort t]})$

Lemma 4.31
$\overset{\downarrow}{=} \text{cc}(\text{flattenf}(\text{mirrorf}\, f))(\text{flattent}(\text{mirrort}\, t))$
IV für Tree/Forest
$\overset{\downarrow}{=} \text{cc}(\text{reverse}(\text{flattenf}\, f))(\text{reverse}(\text{flattent}\, t))$

Anmerkung: hier wird die [Listennotation] als Wald-Notation benutzt.

Übung
Rechte Seite: $\text{reverse}(\text{flattenf}(\text{Cons}\ t\overline{f})) = \text{reverse}(\text{cc}(\text{flattent}\, t)(\text{flattenf}\, f)) \overset{\downarrow}{=}$
$\text{cc}(\text{reverse}(\text{flattenf}\, f))(\text{reverse}(\text{flattent}\, t)) =$ Linke Seite

Lemma 4.30 (Lemma A)**.** $\text{flattenf}(\text{cc}\ fg) = \text{cc}(\text{flattenf}\, f)(\text{flattenf}\, g)$

Lemma 4.31 (Lemma B)**.** $\text{flattenf}\ \texttt{[t]} = \text{flattent}\, t$

□

4.4 Kodatentypen

Daten werden *konstruiert*. Beispiel: $x, l \mapsto \text{Cons}\ x\ l$. Dazu analog werden Kodaten *destruiert* oder *beobachtet*.

Definition 4.32 (Streams)**.** Die Menge der *Streams* S ist definiert als $S = A^\omega = \{\ \underbrace{(a_0, a_1, \dots)}_{\text{unendliche Sequenz}}\ \mid a_i \in A \forall i \geq 0\}$

Außerdem seien die Funktionen *hd* und *tl* definiert wie folgt:

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

und

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

Bemerkung 4.33 (Notation)**.**

```
1 hd: Stream -> A
2 tl: Stream -> Stream
```

Definition 4.34. Ein *Mengenoperator* ist ein Objekt G, welches für Mengen X eine Menge GX definiert.

Bemerkung 4.35 (Recall)**.**
Σ-Algebren $\widehat{=}$ Abbildung $\quad \to M$, wobei F_Σ ein *Mengenoperator* ist (d.h. $F_\Sigma X$ ist Menge für jede Menge X) aus $+$, $\times$ und 1.

z.B. Trees: $\underbrace{1 + M \times M}_{F_{\Sigma_{\text{Tree}}} M} \to M$

Definition 4.36. Dual: Die *G-Koalgebra* ist eine Abbildung $M \to GM$, wobei G ein Mengenoperator ist. G bestehe dabei vorerst nur aus x und konstanten Mengen.

Beispiel 4.37 (Streams)**.**
$GX := A \times X$ (d.h. G bildet jede Menge X ab auf das Kreuzprodukt von A mit dieser; A ist hier konstante Menge.)

$\langle \mathrm{hd}, \mathrm{tl} \rangle : S \to A \times S$ (für $\langle \cdot \rangle$ siehe Definition 4.11).

Erinnerung: Der durch Σ definierte Datentyp ist die initiale Σ-Algebra.

Dual dazu: Der durch G definierte Kodatentyp ist die *final* G-Koalgebra.

Das braucht „Ko-Homomorphismen“, genannt *Koalgebramorphismen*.

Recall: $f : \mathfrak{M} \to \mathfrak{N}$ ist Σ-Homomorphismus $\Leftrightarrow$ folgendes Diagramm kommutiert:

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

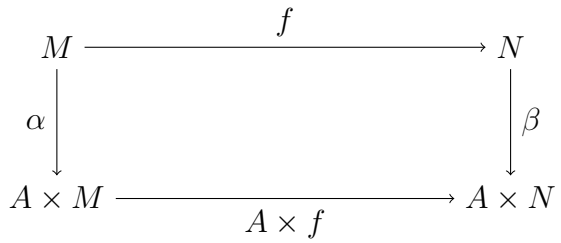
wobei $F_\Sigma f$ definiert ist durch Überladung. z.B. Trees: $F_\Sigma f = 1 + f \times f$.

Dual: $f : \mathfrak{M} \to \mathfrak{N}$ ist G-Koalgebramorphismus $\Leftrightarrow$

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

Beispiel 4.38 (Streams)**.**
verwechsle A und id_A, also ist $Gf = A \times f = id_A \times f$, also $Gf(a, x) = (A \times f)(a, x) = (a, f(x))$.

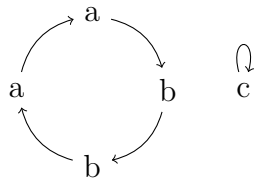
$f : \mathfrak{M} \to \mathfrak{N}$ ist G-Koalgebramorphismus $\Leftrightarrow$



Wenn wir α und β aufspalten in die Komponentenfunktionen, also $\alpha = \langle \mathrm{hd}_{\mathfrak{M}}, \mathrm{tl}_{\mathfrak{M}} \rangle$ und $\beta = \langle \mathrm{hd}_{\mathfrak{N}}, \mathrm{tl}_{\mathfrak{N}} \rangle$:

$\mathrm{hd}_{\mathfrak{N}}(f(x)) = \mathrm{hd}_{\mathfrak{M}}(x)$ und $\mathrm{tl}_{\mathfrak{N}}(f(x)) = f(\mathrm{tl}_{\mathfrak{M}}(x))$

Eine G-Koalgebra ist z.B.:



Definition 4.39. $\mathfrak{N}$ heißt *finale G-Koalgebra*, wenn für jede G-Koalgebra $\mathfrak{M}$ genau ein G-Koalgebramorphismus $\mathfrak{M} \to \mathfrak{N}$ existiert. (Vergleiche auch Definition 4.8)

Satz 4.40. *dual zu initialen Σ-Algebren sind finale G-Koalgebren eindeutig bis auf Isomorphismus.*

Satz 4.41. *Mit hd, tl und S wie in Definition 4.32, und $GX = A \times X$ ist S eine finale G-Koalgebra.*

Beweis. Sei $\mathfrak{M}$ G-Koalgebra (mit $\mathfrak{M}[\![\mathrm{hd}]\!] = \mathrm{hd}_{\mathfrak{N}}$ etc.).

Für ein Element (denke: „Zustand“) $x \in M$ setze $f(x) := (a_0, a_1, \dots)$ mit $a_i := \mathrm{hd}_{\mathfrak{M}}(\mathrm{tl}^i_{\mathfrak{M}}(x))$.

Zu zeigen sind:

1. z.Z.: f ist G-Koalgebramorphismus.

   $\mathrm{hd}(f(x)) = a_0 = \mathrm{hd}_{\mathfrak{M}}(x)$. ✓.

   $\mathrm{tl}(f(x)) = (a_1, a_2, \dots) =: (b_0, b_1, \dots)$ mit $b_i := a_{i+1} = \mathrm{hd}_{\mathfrak{M}}(\mathrm{tl}^{i+1}_{\mathfrak{M}}(x)) = \mathrm{hd}_{\mathfrak{M}}(\mathrm{tl}^i_{\mathfrak{M}}(\mathrm{tl}_{\mathfrak{M}}(x)))$.

   also $\mathrm{tl}(f(x)) = f(\mathrm{tl}_{\mathfrak{M}}(x))$. ✓.

2. z.Z.: f ist eindeutig.

Sei $g : \mathfrak{M} \to S$ ein G-Koalgebramorphiemus. Zeige für alle Zustände x von M, also $\forall x \in M$, dass gilt: $g(x)_i = f(x)_i$ (der Index i bedeutet „i-te Position im Stream“).

per Induktion über i:

- $i = 0$:

$$g(x)_0 = \mathrm{hd}(g(x)) \overset{g \text{ ist Koalgebramorphismus}}{\overset{\downarrow}{=}} \mathrm{hd}_{\mathfrak{M}}(x) = f(x) \quad \checkmark$$

- $i \to i + 1$:

$$g(x)_{i+1} = \mathrm{tl}(f(x))_i \overset{g \text{ ist Koalgebramorphismus}}{\overset{\downarrow}{=}} g(\mathrm{tl}_{\mathfrak{M}}(x))_i \overset{IV}{=} f(\mathrm{tl}_{\mathfrak{M}}(x))_i \overset{f \text{ ist Koalgebramorphismus}}{\overset{\downarrow}{=}} f(x)_{i_1} \quad \checkmark$$

□

Definition 4.42. d.h. für $h : M \to A$, $t : M \to M$ wird unfold ht definiert per

$$\mathrm{hd}(\text{unfold } h\ t\ x\ ) = h\ x$$
$$\mathrm{tl}(\text{unfold } h\ t\ x\ ) = \text{unfold } h\ t\ (t\ x)$$

Beispiel 4.43 (ones)**.**
ones $: 1 \to S$, $\mathrm{hd}\,\text{ones} = 1$, $\mathrm{tl}\,\text{ones} = \text{ones}$.

ones $=$ unfold $(\lambda x.1) id_1$

Beispiel 4.44 (blink)**.**
Wir versuchen, ein blink zu definieren, so dass es blink $x(a_0, a_1, \dots) = (x, a_0, x, a_1, x, a_2, \dots)$ erfüllt.

$\mathrm{hd}(\text{blink } x\ s) = x$ und $\mathrm{tl}(\text{blink } x\ s) =?$

wir brauchen also eine Fallunterscheidung:

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

Schreibe blink $0 = f$, blink $1 = g$

$\mathrm{hd}(f\ x\ s) = x$
$\mathrm{tl}(f\ x\ s) = g\ x\ s$
$\mathrm{hd}(g\ x\ s) = \mathrm{hd}\, s$
$\mathrm{tl}(g\ x\ s) = f\ x\ (\mathrm{tl}\, s)$

4.5 Koinduktion

Definition von *even*, *odd*:
$\mathrm{hd}(\mathrm{even}\, s) = \mathrm{hd}\, s$
$\mathrm{hd}(\mathrm{odd}\, s) = \mathrm{hd}(\mathrm{tl}\, s)$
$\mathrm{tl}(\mathrm{even}\, s) = \mathrm{odd}(\mathrm{tl}\, s) = \mathrm{even}(\mathrm{tl}(\mathrm{tl}\, s))$
$\mathrm{tl}(\mathrm{odd}\, s) = \mathrm{odd}(\mathrm{tl}(\mathrm{tl}\, s))$

Definition 4.45. $\mathrm{R} \subseteq S \times S$ heißt *Bisimulation*, wenn für alle $(s,t) \in \mathrm{R}$ gilt:

1. $\mathrm{hd}\, s = \mathrm{hd}\, t$
2. $(\mathrm{tl}\, s)\, \mathrm{R}\, (\mathrm{tl}\, t)$

Satz 4.46 (Koinduktion)**.** *Wenn* R *Bisimulation ist, und* $\underbrace{s\, \mathrm{R}\, t \text{ dann } s = t}_{\text{also } \mathrm{R} \subseteq id}$

Beweis. Sei $s = (x_0, x_1, \dots), t = (y_0, y_1, \dots)$. Zeige $(*)$ $(x_i, x_{i+1}, \dots)\, \mathrm{R}\, (y_i, y_{i+1}, \dots)$ per Induktion über $i \geq 0$.

- $i = 0$: ✓ nach Voraussetzung des Satzes
- $i \to i+1$. Nach Induktionsvoraussetzung ist $(x_i, x_{i+1}, \dots)\, \mathrm{R}\, (y_i, y_{i+1}, \dots)$. Da R Bisimulation ist, folgt $\mathrm{tl}(x_i, x_{i+1}, \dots)\, \mathrm{R}\, \mathrm{tl}(y_i, y_{i+1}, \dots)$, also $(x_{i+1}, x_{i+2}, \dots)\, \mathrm{R}\, (y_{i+1}, y_{i+2}, \dots$ ✓

$(*) \overset{\text{R ist Bisimulation}}{\Rightarrow} \underbrace{\mathrm{hd}(x_i, x_{i+1}, \dots)}_{=x_i} = \underbrace{(y_i, y_{i+1}, \dots)}_{=y_i}$, also $s = t$ □

Bemerkung 4.47.
Die Umkehrung ist klar. (*Vollständigkeit*: Wenn $s = t$, dann existiert eine Bisimulation R mit $s\, \mathrm{R}\, t$, nämlich $\mathrm{R} = id$.)

Beispiel 4.48.
Wir überprüfen, ob $\mathrm{R} = \left\{ \Big(\mathrm{odd}(f\ x\ s), s \Big) \mid x \in A, s \in S \right\}$ eine Bisimulation ist.

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

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

Beispiel 4.49.
Anschauung: $\mathrm{zip}(x_0, \dots)(y_0, \dots) = (x_0, y_0, x_1, y_1, \dots)$

$\mathrm{hd}(\mathrm{zip}\, s\ t) = \mathrm{hd}\, s$
$\mathrm{tl}(\mathrm{zip}\, s\ t) = \mathrm{zip}\, t\ (\mathrm{tl}\, s)$

Behauptung: $\mathrm{zip}(\mathrm{even}\, s)(\mathrm{odd}\, s) = s$

Beweis. Ist $R = \left\{ \left(\mathrm{zip}(\mathrm{even}\, s)(\mathrm{odd}\, s), s \right) \mid s \in S \right\}$ eine Bisimulation?

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

Die Relation R scheint zu klein gewählt. Setze also
$\mathrm{R}' := \mathrm{R} \cup \left\{ \left(\mathrm{zip}(\mathrm{odd}\, s)(\mathrm{even}(\mathrm{tl}(\mathrm{tl}\, s))), \mathrm{tl}\, s \right) \mid s \in S \right\}$

$\mathrm{hd}(\mathrm{zip}(\mathrm{odd}\, s)(\ldots)) = \mathrm{hd}(\mathrm{odd}\, s) = \mathrm{hd}(\mathrm{tl}\, s))$ ✓

$\mathrm{tl}(\mathrm{zip}(\mathrm{odd}\, s)(\mathrm{even}(\mathrm{tl}(\mathrm{tl}\, s))) = \mathrm{zip}(\mathrm{even}(\mathrm{tl}(\mathrm{tl}\, s)))(\mathrm{tl}(\mathrm{odd}\, s))$
$= \mathrm{zip}(\mathrm{even}(\mathrm{tl}(\mathrm{tl}\, s)))(\mathrm{odd}(\mathrm{tl}(\mathrm{tl}\, s)))\quad \mathrm{R}'\quad \mathrm{tl}(\mathrm{tl}\, s)$ □

4.6 Kodatentypen mit Alternativen

Beispiel:

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

Das definiert die finale G-Koalgebra für $GX = 1 + A \times X$ (mit $A = [\![a]\!]$).

also $\alpha : N \to 1 + A \times N$

Das gibt disjunkte Zerlegung

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

Dann eben

$$\mathrm{end} : \mathrm{dead} \to 1$$
$$\langle \mathrm{hd}, \mathrm{tl} \rangle : \mathrm{alive} \to A \times N$$

Definition 4.50. Allgemein: Für $GX = \sum_{i=1}^{n} A_i \times X^{K_i}$ beschreibe die G-Koalgebra durch die *Signatur* Σ aus

- Alternativen $d_1, \ldots, d_n$ (oben: $d_1 = \text{dead}, d_2 = \text{alive}$)
- für $i = 1, \ldots, n$ je k_{i+1} *Observer*

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

  (Hinweis: $[\![a_i]\!] = A_i$, $j = 1, \ldots, k_i$)

  (Beispiel: *kein* t_{0j}, $t_{11} = \text{tl}$, $h_1 = \text{hd}$)

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

- Träger N
- disjunkter Zerlegung $N = \dot{\bigcup}_{i=1}^{n} \mathfrak{N}[\![d_i]\!]$
- $\mathfrak{N}[\![t_{ij}]\!] : \mathfrak{N}[\![d_i]\!] \to N$
- $\mathfrak{N}[\![h_i]\!] : \mathfrak{N}[\![d_i]\!] \to A_i$

Definition 4.51. Σ-Morphismen $f : \mathfrak{M} \to \mathfrak{N}$:

hmm ist das wirklich eine Definition?

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

Finale Σ-Koalgebra: unendliche Bäume

- Knoten markiert mit Alternative d_i und $x \in A_i$
- Dann k_i Kinder (eines pro t_{ij})

z.B. `IList`: endliche oder unendliche Listen über A.

Beispiel 4.52 (takeUntil)**.**
$\text{takeUntil} : a \times \text{IList}\, a \to \text{IList}\, a$
$\text{dead}_{\text{takeUntil}}(x, s@alive) \Leftrightarrow \text{hd}\, s = x$
$\text{dead}_{\text{takeUntil}}(x, s@dead)$

$$\text{alive}_{\text{takeUntil}}(x, s) \to \begin{cases} \text{hd}(\text{takeUntil}\, x\ s) = \text{hd}\, s \\ \text{tl}(\text{takeUntil}\, x\ s) = \text{takeUntil}\, x\ (\text{tl}\, s) \end{cases}$$

```
codata Terrain a where
        objects: Terrain a -> List a
        deadend: Terrain a @ stuck -> ()
        left:    Terrain a @ junction -> Terrain a
        right:   Terrain a @ junction -> Terrain a
```

Wenn man das als Koalgebra betrachtet: $T_a X = \overset{\text{List a}}{\underset{\downarrow}{a^*}} \times (\overset{\text{stuck}}{\underset{\downarrow}{1}} + X^2)$

Die finale Koalgebra für T_a enthält also alle *strikten* Binärbäume (d.h. ein Knoten hat entweder keinen oder zwei Nachfolger), sodass:

- jeder Knoten wird durch eine `List a` benannt
- unendliche Verzweignug ist zulässig.

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 das 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 4.53. $R \subseteq \text{Terrain} \times \text{Terrain}$ heisst (Terrain-)bisimulation, wenn für $x \,\mathrm{R}\, y$ gilt:

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

Satz 4.54. *Sei* R *eine Terrain-Bisimulation ist und es gelte* $x \,\mathrm{R}\, y$. *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.

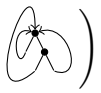
Proposition 4.55. $Id := \{(x, x) \mid x \in \text{Terrain}\}$ *ist eine Bisimulation.*

Proposition 4.56. *Wenn S und R Bisimulationen sind, dann ist $S \cup R$ auch eine Bisimulation.*

Beweisprinzip: Wenn wir wissen, dass S eine Bisimulation ist, und wir zeigen wollen, dass $R \cup S$ auch Bisimulation ist, dann müssen wir nur Bedingungen 1-3 (siehe Definition 4.53) auf Paaren $(x, y) \in R \subseteq R \cup S$ zeigen.

$R := \{\dots\} \cup Id$

$\text{loop}\,\text{Terrain}\,a \to \text{Terrain}\,a$

$\text{loop}\left(\right) \mapsto \left(\right)$ 

Idee: Hilfsfunktion loopBack, die einen Baum t so verändert, dass alle „stuck“-Knoten so aussehen und sich so verhalten wie s (Voraussetzung: $s@junction$!):

$\text{loop}\,s = \text{loopBack}\ \ s\ s$

$\text{objects}(\text{loopBack}\ \ t@junction\ s) = \text{objects}\,t$
$\text{objects}(\text{loopBack}\ \ t@stuck\ s) = \text{objects}\,s$

$\text{left}(\text{loopBack}\ \ t@junctions) = \text{loopBack}\ (\text{left}\,t)\ s$
$\text{left}(\text{loopBack}\ \ t@stucks) = \text{loopBack}\ (\text{left}\,s)\ s$

$\text{right}(\text{loopBack}\ \ t@junctions) = \text{loopBack}\ (\text{right}\,t)\ s$
$\text{right}(\text{loopBack}\ \ t@stucks) = \text{loopBack}\ (\text{right}\,s)\ s$

Zu zeigen: $\text{loop}(\text{loop}\,s) = \text{loop}\,s$

Definiere die Relation $R := \Big\{\Big(\,\text{loop}(\text{loop}\,s), \text{loop}\,s\Big) \mid s \in \text{Terrain}\,@junction\Big\}$.

Wir wollen zeigen, dass R eine Bisimulation ist. Betrachte $\text{loop}(\text{loop}\,s)\,\text{R}\,\text{loop}\,S$, dann haben wir:

1. $\text{objects}\Big(\,\text{loop}(\text{loop}\,s)\Big) =$
   $\text{objects}\Big(\,\text{loopBack}(\overset{@junction}{\overset{\downarrow}{\text{loop}}}\,s)(\text{loop}\,s)\Big) =$
   $\text{objects}(\text{loop}\,s)$

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

3. Offensichtlich ist $(\text{loop}\,s)@junction$, also bleibt nur noch zu prüfen:

a) zu zeigen: $\text{left}(\text{loop}(\text{loop}\, s))\, \mathrm{R}\, \text{left}(\text{loop}\, s)$

$\text{left}(\text{loop}(\text{loop}\, s)) = \text{left}\left(\text{loopBack}(\text{loop}\, s)(\text{loop}\, s)\right) =$
$\text{loopBack}(\text{left}(\text{loop}\, s))(\text{loop}\, s) =$
$\text{loopBack}(\text{left}(\text{loopBack}\, s\ s)(\text{loop}\, s)$

An diesem Punkt erweitert man die ursprüngliche Relation R um
$\left\{\left(\text{loopBack}(\text{loopBack}\, t\ s)(\text{loop}\, s), \text{loopBack}\, t\ s\right) \mid t \in \text{Terrain}, s \in \text{Terrain}\, @junction\right\}$.

$\text{loopBack}(\text{loopBack}(\text{left}\, s)s)(\text{loop}\, s)\, \mathrm{R}\, \text{loopBack}(\text{left}\, s)$

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

b) symmetrisch für right(...).

Wir nehmen $\text{loopBack}(\text{loopBack}\, t\ s)(\text{loop}\, s)\, \mathrm{R}\, \text{loopBack}\, t\ s$ an:

a) $\text{objects}\left(\text{loopBack}(\text{loopBack}\, t\ s)(\text{loop}\, s)\right) =$
$\text{objects}(\text{loopBack}\, t\ s)$ ✓

b) trivial ✓

c) i. $\text{left}\left(\text{loopBack}(\text{loopBack}\, t\ s)(\text{loop}\, s)\right) =$
$\text{loopBack}\left(\text{left}(\text{loopBack}\, t\ s)\right)(\text{loop}\, s) =$
$\begin{cases}\text{loopBack}\left(\text{loopBack}(\text{left}\, s)s\right)(\text{loop}\, s) & \text{falls } t@stuck \\ \text{loopBack}\left(\text{loopBack}(\text{left}\, t)s\right)(\text{loop}\, s) & \text{falls } t@junction\end{cases}$

nicht über mismatched Klammern wundern.

$\text{left}(\text{loopBack}\, t\ s) = \begin{cases}\text{loopBack}(\text{left}\, s)s & \text{falls } t@stuck \\ \text{loopBack}(\text{left}\, t)s & \text{falls } t@junction\end{cases}$ also kommutieren diese drei Fälle.

□

Im Folgenden wollen wir einen Kodatentyp *InfTerrain* definieren, der sowohl vom bereits bekannten Typparameter a abhängt, zusätzlich aber auch von einem „Richtungs-enum“ *dir*:

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

$$T_{dir,a}X = \underset{\substack{\uparrow \\ \text{List a}}}{a^*} \times X^{dir}$$

R ist also eine InfTerrain-Bisimulation, wenn für $x \,\mathrm{R}\, y$ gilt:

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

4.7 Polymorphie und System-F

Polymorphie ist bereits aus Programmiersprachen wie C++ bekannt. Beispiele hierfür sind die Überladung des `operator+()`, sodass er sowohl auf `int`s als auch auf `string`s angewandt werden kann.

Ein weiteres Beispiel ist die Implementierung von Interfaces in Java:

```
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 }
```

Doch auch folgender Code zeigt eine Form der Polymorphie:

```
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)
```

Es gibt zwei Arten von Polymorphie:

- Parametrische Polymorphie: Eine einzelne Codepassage erhält einen generischen Typ. Das Verhalten ist *gleichförmig* auf allen Instanzen.

  Beispiele: Haskell wie oben. Bis zu einem gewissen Grad kann man auch Java-Generics hierzu zählen, wenn man aber auf den `instanceof`-Operator und andere Funktionen verzichtet, die eine unterschiedliche Behandlung unterschiedlicher Typen ermöglichen.

- Ad-hoc Polymorphie: Jede Instanz kann sich anders verhalten.

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

Generics in Java sind nicht eindeutig zuzuordnen, wegen `instanceof`.

Ad-Hoc-Polymorphie ist letztendlich nur ein nettes Bequemlichkeitsfeature, während parametrische Polymorphie weitaus *fundamentaler* und mächtiger ist.

Als nächstes: Ein polymorphes Typsystem für den λ-Kalkül.

Bemerkung 4.57 (Erinnerung)**.**
Die drei Typisierungsregeln des einfach getypten λ-Kalküls (STLC) sind:

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

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

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

Polymorphie in der Praxis, ein Beispiel:

```
1 id :: a -> a
2 id = \x -> x
3
4 silly :: Int
5 silly = (\x y -> x) (id True) (id 42)
```

Können wir `silly` einen Typ im einfach getypten λ-Kalkül zuweisen?

$\lambda x.x : a \to a$
$\Gamma = \{\text{True} : \text{Bool}, \text{False} : \text{Bool}, 0 : \text{Int}, 1 : \text{Int}, \ldots, 42 : \text{Int}, \ldots, \text{id} : a \to a\}$

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

Es gibt zwei Versionen des System F: von Curry und von Church.

Definition 4.58 (System-F nach Curry)**.**

- Typen: $\alpha, \beta ::= a \mid \alpha \to \beta \mid \forall a.\alpha$ (wobei a Typvariable ist)
- Terme: $s, t ::= x \mid st \mid \lambda x.s$ (wobei x Termvariable ist)
- Typregeln (die ersten drei sind dieselben wie beim einfach getypten λ-Kalkül):

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

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

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

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

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

Anmerkung: $FV(\Gamma)$ bezeichnet die freien Variablen von Γ und beschreibt ganz analog alle Typvariablen, die nicht durch einen $\forall$-Quantor gebunden sind.

Bemerkung 4.59.
Im System F lässt sich `silly` typisieren!

Satz 4.60 (Preservation)**.** *Wenn* $\Gamma \vdash s : \alpha$ *und* $s \to_\beta t$*, dann* $\Gamma \vdash t : \alpha$

Satz 4.61 (Normalisierung, Girard)**.** *Wenn* $\Gamma \vdash s : \alpha$*, dann ist* s *stark normalisierend*

Tatsächlich kann jede totale berechenbare Funktion, die „SO“-Arithmetik nutzt, in System F geschrieben werden!

- Alle primitiv rekursiven Funktionen
- Sogar die Ackermannfunktion
- Intuition: Ein Compiler ja, ein Interpreter nein.

4.7.1 Church-Kodierung in System F

Definition 4.62 (Natürliche Zahlen)**.** Analog zum einfach getypten λ-Kalkül kann die *Church-Kodierung* von natürlichen Zahlen im System F definiert werden wie folgt:

- $\mathbb{N} := \forall a \,.\, (a \to a) \to a \to a$
- zero : $\mathbb{N}$
  zero : $\lambda f a \,.\, a$
- succ : $\mathbb{N} \to \mathbb{N}$
  succ = $\lambda n f a \,.\, f(n f a)$

- $\text{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
  $\text{add} = \lambda nm \,.\, n\,\text{succ}\,m$

Definition 4.63 (Paare)**.**
$(a \times b) := \forall r \,.\, (a \to b \to r) \to r$

$\text{pair} : \forall ab \,.\, a \to b \to (a \times b)$
$\text{pair} = \lambda xyf \,.\, fxy$

$\text{fst} : \forall ab \,.\, (a \times b) \to a$
$\text{fst} = \lambda p \,.\, p(\lambda xy \,.\, x)$

$\text{snd} : \forall ab \,.\, (a \times b) \to b$
$\text{snd} = \lambda p \,.\, p(\lambda xy \,.\, y)$

Definition 4.64 (Summen)**.**
$(a + b) := \forall r \,.\, (a \to r) \to (b \to r) \to r$

$\text{injL} : \forall ab \,.\, a \to (a + b)$
$\text{injL} = \lambda \overset{a}{\downarrow} xfg \,.\, fx$

$\text{injR} : \forall ab \,.\, b \to (a + b)$
$\text{injR} = \lambda \overset{b}{\downarrow} yfg \,.\, gy$

$\text{case} : \forall abs \,.\, (a \to s) \to (b \to s) \to (a + b) \to s$
$\text{case} = \lambda fgs \,.\, sfg$

```
1 data Either a b where
2         injL: a -> Either a b
3         injR: b -> Either a b
4
5 f: Either Int Bool -> Int
6 f (injL n) = n                          -- Typ: Int -> Int
7 f (injR b) = if b then 1 else 0         -- Typ: Bool -> Int
```

Definition 4.65 (Listen)**.**
$\text{List}\,a := \forall r \,.\, r \to (a \to r \to r) \to r$

$\text{Nil} : \forall a \,.\, \text{List}\,a$
$\text{Nil} = \lambda uf \,.\, u$

$\text{Cons} : \forall a \,.\, a \to \text{List}\,a \to \text{List}\,a$
$\text{Cons} = \lambda xluf \,.\, fx(luf)$

$\text{length} : \forall a \,.\, \text{List}\,a \to \mathbb{N}$
$\text{length} = \lambda l \,.\, l\,\text{zero}(\lambda xr \,.\, \text{succ}\,r)$

Typinferenz im System F ist *unentscheidbar*. Zwei Ansätze zur Lösung:

- Einschränken der Sprache zu etwas weniger mächtigem ($\rightarrow$ ML-Polymorphie)
- Ändern des Problems, von der *Typinferenz* zur *Typüberprüfung*

4.8 ML-Polymorphie

Die Idee hinter der ML-Polymorphie ist:

- $\forall$ ist nur toplevel (d.h. es darf nicht im linken Argument von $\rightarrow$ auftauchen).

  Beispielsweise ist folgendes *nicht* mehr erlaubt, da der $\forall$-Quantor unterhalb des unterstrichenen $\rightarrow$ anzusiedeln ist:

$$\forall a.\Big((a \to a) \to a \to a\Big) \underline{\to} \forall a.(a \to a) \to a \to a$$

- Mehrfachinstanziierung polymorpher Funktionen ist nur per *let* erlaubt („let-Polymorphie“):

$$\text{let id} = \lambda x.x \text{ in } \underbrace{\text{id}}_{(a\to a)\to a\to a} (\underbrace{\text{id}}_{a\to a})$$

- Entscheidbare Typinferenz

Definition 4.66.

- Typen: $\alpha, \beta ::= a \mid \alpha \to \beta$
- Typschemata: $S ::= \forall a_1, \ldots, a_k : \alpha$
- Terme: $t, s ::= x \mid t\, s \mid \lambda x.t \mid \text{let } x = t \text{ in } s$
- Kontexte: $\Gamma = (x_1 : S_1, \ldots, x_n : S_n)$
- Typisierung: $\Gamma \vdash t : \alpha$ (lies: t hat das Typschema $\underbrace{\forall a_1, \ldots, a_k.\alpha}_{:=Cl(\Gamma,\alpha)}$)

  mit $a_1, \ldots, a_k \in FV(\alpha) \setminus \underbrace{FV(\Gamma)}_{=\cup FV(S_i)}$

Beispiel 4.67.
$\vdash \lambda x.x : a \to a, \quad Cl\Big((), a \to a\Big) = \forall a.a \to a$

Definition 4.68. Die Typisierungsregeln sind:

- $(\forall_e)\ \dfrac{}{\Gamma\ \vdash\ x : \alpha[a_1 \mapsto \beta_1, \ldots, a_k \mapsto \beta_k]}$

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

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

- $(\text{let})\ \dfrac{\Gamma\ \vdash\ t : \alpha \qquad \Gamma,\, x : Cl(\Gamma, \alpha)\ \vdash\ s : \beta}{\Gamma\ \vdash\ \text{let}\, x = t\ \text{in}\ s : \beta}$

Beispiel 4.69.

$$\dfrac{\alpha = a \to a\ \dfrac{\ldots}{\vdash \lambda x.x : \alpha} \qquad \dfrac{\text{id} : \forall a.a \to a\ \vdash\ \text{id} : \gamma \to \beta \qquad \text{id} : \forall a.a \to a\ \vdash\ \text{id} : \gamma}{\text{id} : \forall a.a \to a\ \vdash\ \text{id}\,\text{id} : \beta}}{\vdash\ \text{let}\,\text{id} = \lambda x.x\ \text{in}\ \ \text{id}\,\text{id} : \beta}$$

z.B. mit $\gamma = b \to b, \beta = b \to b$

Intuitiv denkt man, dass die untere Zeile im Prooftree $= \underbrace{(\lambda\,\text{id}\,.\,\text{id}\,\text{id})}\ \lambda x.x$ ist. Der geklammerte Ausdruck ist aber nicht typisierbar.

Algorithmus 4.70 (Algorithmus W)**.**
Definition des Algorithmus durch ein *Regelsystem* für $\Gamma \vdash t : \sigma, \alpha$ mit $\Gamma \vdash t : \sigma, \alpha \Rightarrow$ σ ist allgemeine Lösung für $\Gamma\sigma \vdash t : \alpha\sigma$.

- Habe x den Typ $\forall a_1, \ldots, a_k.\alpha$ im Kontext Γ, d.h.
  $x : \forall a_1, \ldots, a_k.\alpha \quad \in \quad \Gamma$.

  $$\dfrac{}{\Gamma\ \vdash\ x : \underset{\substack{\uparrow \\ \text{die Substitution, die nichts tut}}}{[\,]}, \alpha[a_1 \mapsto b_1, \ldots, a_k \mapsto b_k]}$$

  $b_1, \ldots, b_k$ frisch.

- $\dfrac{\Gamma\ \vdash\ t : \sigma, \alpha \qquad \Gamma\ \vdash\ s : \tau, \beta}{\Gamma\ \vdash\ ts : \sigma\tau\vartheta}$

  wobei $\vartheta = mgu\Big(\sigma(\alpha), \tau(\beta) \to a\Big)$

- $$\frac{\Gamma,\, x:\alpha \;\vdash\; t:\sigma,\alpha}{\Gamma \;\vdash\; \lambda x.t:\sigma,\sigma(a)\to\alpha}$$

wobei a frisch ist.

- $$\frac{\Gamma \;\vdash\; t:\sigma,\alpha \qquad \Gamma,\, x:Cl(\Gamma,\alpha) \;\vdash\; s:\tau,\beta}{\Gamma \;\vdash\; \mathrm{let}\, x = t \;\mathrm{in}\; s:\sigma\tau,\beta}$$

Beispiel 4.71.

$$\dfrac{\dfrac{x:a \;\vdash\; x:[\,],a}{\vdash \lambda x.x:\sigma,\alpha} \qquad \overbrace{\mathrm{id}:\forall a.a\to a}^{S}\,\mathrm{id}\,\mathrm{id}:\tau,\beta}{\vdash \mathrm{let}\,\mathrm{id} = \lambda x.x \;\mathrm{in}\; \mathrm{id}\,\mathrm{id}} \qquad \dfrac{\mathrm{id}:S \;\vdash\; \overset{=[\,]}{\overset{\downarrow}{\tau}},\gamma\to c \qquad \mathrm{id}:S \;\vdash\; \mathrm{id}:\overset{=[\,]}{\overset{\downarrow}{\vartheta}},\gamma}{\underset{(d\to d)\to(d\to d)}{\uparrow}\qquad\qquad \underset{=d\to d}{\uparrow}}$$

$\sigma = [\,],\ \alpha = a\to a$

Beispiel 4.72 (Dasselbe nochmal in sauberer)**.**

$$\dfrac{\dfrac{x:a \;\vdash\; x:[\,],a}{\vdash \lambda x.x:\sigma,\alpha} \qquad \overbrace{\mathrm{id}:\forall a.a\to a}^{S}\,\mathrm{id}\,\mathrm{id}:\tau,\beta}{\vdash \mathrm{let}\,\mathrm{id} = \lambda x.x \;\mathrm{in}\; \mathrm{id}\,\mathrm{id}} \qquad \dfrac{\mathrm{id}:S \;\vdash\; \mathrm{id}:[\,],b\to b \qquad \mathrm{id}:S \;\vdash\; \mathrm{id}:[\,],c\to c}{}$$

$\sigma = [\,],\ \alpha = a\to a$

$\tau = mgu(b\to b,(c\to c)\to d) = [b\mapsto(c\to c), d\mapsto(c\to c)]$, und $\beta = \tau(d) = c\to c$

4.9 Curry vs. Church

Curry ist typisierbar, während Church getypt ist.

Bemerkung 4.73 (Erinnerung)**.**
$\lambda \to$ nach Church hat

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

mit Typregeln:

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

Definition 4.74 ($\lambda 2$-Church)**.** $t, s ::= x \mid ts \mid \lambda x : a.t \mid \Lambda a.t \mid t\alpha$

mit $(\Lambda a.t)\alpha \to_\beta t[a \mapsto \alpha]$

Typregeln:

- $(\forall_e) \dfrac{\Gamma \;\vdash\; t : \forall a.\alpha}{\Gamma \;\vdash\; t\beta : \alpha[a \mapsto \beta]}$

- $(\forall_i) \dfrac{\Gamma \;\vdash\; t : \alpha}{\Gamma \;\vdash\; \Lambda a.t : \forall a.\alpha}$

Beispiel 4.75.

1. $\vdash\; \Lambda b.\lambda x : (\forall a.a).xb : \forall b.(\forall a.a) \to b$
2. $\vdash\; \Lambda b.\lambda x : (\forall a.a)x(\forall a.a \to b)x : \forall b.\Big((\forall a.a) \to b\Big)$

Diese beiden Beispiele demonstrieren das Prinzip *Ex falso quod libet*, d.h. aus einer falschen Annahme kann beliebiges (unsinniges) gefolgt werden.

5 Reguläre Ausdrücke

Literatur hierzu: „*Lecture Notes on Regular Languages and Finite Automata*“, Andrew Pitts, Cambridge, 2013

ggf Literatur-section

5.1 Recall: Nichtdeterministischer endlicher Automat

Definition 5.1. Ein *nichtdeterministischer endlicher Automat mit ϵ-Transitionen* (NFA^ϵ) ist ein Tupel

$$A = (Q, \Sigma, \Delta, s, F)$$

mit:

- einer Menge von *Zuständen* Q
- einem *Alphabet* Σ
- einer *Transitionsabbildung* $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to P(Q)$
  ↑ Potenzmenge

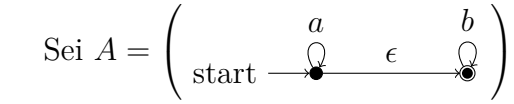
  $q \xrightarrow{a} q' :\Leftrightarrow q' \in \Delta(q, a)$
- einem *Anfangszustand* $s \in Q$
- einer Menge der akzeptierenden/finalen Zustände $F \subseteq Q$

Definition 5.2. Definiere $q \overset{u}{\Longrightarrow} q'$ für $u \in \Sigma^*$ induktiv per

- $$\frac{}{q \overset{\epsilon}{\Longrightarrow} q}$$
- $$\frac{q \overset{u}{\Longrightarrow} q' \qquad q' \xrightarrow{\epsilon} q''}{q \overset{u}{\Longrightarrow} q''}$$
- $$\frac{q \overset{u}{\Longrightarrow} q' \qquad q' \xrightarrow{a} q''}{q \overset{ua}{\Longrightarrow} q''}$$

Dann $L(A) = \{u \in \Sigma^* \mid \exists q \in F \text{ mit } s \overset{u}{\Longrightarrow} q\} \to$ reguläre Sprachen.

Beispiel 5.3.

Sei $A =$ 

$L(A) = \{a^n b^k \mid n, k \geq 0\}$

Definition 5.4. *A* NFA $\Leftrightarrow \Delta(q, \epsilon) = \{\}$ immer.

NFA A ist *deterministisch* (DFA), wenn $|\Delta(q, a)| = 1$ immer. Dann $\Delta(q, a) =: \{\delta(q, a)\}$

Lemma 5.5 (Potenzmengenkonstruktion)**.** *Zu jedem NFA$^\epsilon$ A existiert ein DFA B mit* $L(B) = L(A)$

5.2 Reguläre Ausdrücke

Definition 5.6. *Reguläre Ausdrücke* sind von der Form:

$$r, s ::= 1 \mid 0 \mid r + s \mid rs \mid r^* \mid a$$

für $a \in \Sigma$

Semantik: $L(r)$ ist eine rekursiv definierte Sprache:

- $L(a) = \{a\}$
- $L(1) = \{\epsilon\}$
- $L(0) = \{\}$
- $L(r + s) = L(r) \cup L(s)$
- $L(rs) = \{uv \mid u \in L(r), v \in L(s)\}$
- $L(r^*) = \{u_1, \ldots, u_n \mid n \geq 0, u_i \in L(r) \forall i\}$

Beispiel 5.7.
Der reguläre Ausdruck für „höchstens zwei a nacheinander“ mit $\Sigma = \{a, b\}$ ist: $(b + ab + aab)^* + (1 + a + aa)$

In `grep`-Notation: `(b|ab|aab)*(|a|aa)`

Satz 5.8 (Satz von Kleene)**.** *Eine Sprache L ist regulär, genau dann wenn es einen regulären Ausdruck r gibt mit $L = L(r)$*

Beweis. „$\Leftarrow$“: Per Induktion über r:

- falls $r = 0$:

  start $\longrightarrow$●

- falls $r = 1$:

  start $\longrightarrow$ ◉

- falls $r = a$:

  start $\longrightarrow$ ● $\xrightarrow{a}$ ◉

- für $r + s$: bilder dazu malen

  Initialer Zustand kann über ϵ-Übergang entweder auf den Startzustand des r akzeptierenden Automaten, oder auf den Startzustand des s akzeptierenden Automaten kommen.

- für rs:

  Initialer Zustand ist initialer Zustand des r-Automaten; akzeptierende Zustände des r-Automaten gehen über ϵ-Übergang in den Startzustand des s- Automaten. Akzeptierende Zustände sind die akzeptierenden Zustände des s- Automaten.

- für r^*

  Künstlicher akzeptierender Startzustand mit ϵ-Übergang in r-Automaten; akzeptierende Zustände des s-Automaten können per ϵ-Übergang wieder in den künstlichen Startzustand gelangen.

„⇒“: Sei $A = (Q, \Sigma, \delta, s, F)$ ein deterministischer endlicher Automat.

Zeige $(*) : \forall q, q' \in Q, R \subseteq Q$ gibt es einen regulären Ausdruck $r^R_{q,q'}$ mit $L(r^R_{q,q'}) = \{u \in \Sigma^* \mid q \xrightarrow{u} q'$ 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 $|R|$:

- sei $|R| = 0$
  - Fall 1: $q \neq q'$: $r^\emptyset_{q,q'} = \sum_{q \xrightarrow{a} q'} a$
  - Fall 2: $q = q'$: $r^\emptyset_{q,q'} = 1 + \sum_{q \xrightarrow{a} q'} a$
- sei $|R| > 0$: Wähle dann $q_0 \in R$, $R_0 := R \setminus \{q_0\}$. Wir haben nach Induktionsvoraussetzung schon alle $r^{R_0}_{q,q'}$.

zerteile einen Weg in jedem Punkt, wenn er über q_0 läuft, und konkateniere diese Teilwege

Setze also $r^R_{q,q'} := r^{R_0}_{q,q'} + r^{R_0}_{q,q_0} \left(r^{R_0}_{q_0,q_0}\right)^* r^{R_0}_{q_0,q'}$.

wir laufen von q nach q' ohne Umweg über q_0

□

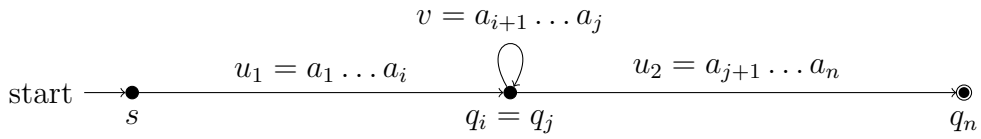
Bemerkung 5.9.
Anwendung: reguläre Ausdrücke sind abgeschlossen unter $\cap$ und Komplement.

Satz 5.10 (Pumping-Lemma)**.** *Wenn eine Sprache L regulär ist, dann $\exists l \geq 1$, sodass $\forall w \in L$ mit $|w| \geq l$ gilt: $\exists u_1, v, u_2$ mit $w = u_1 v u_2$ und $|v| \geq 1 \wedge |u_1 v| \leq l \wedge u_1 v^* u_2 \subseteq L$.*

Kontraposition: L ist nicht *regulär, wenn $\forall l \geq 1 \exists w \in L$ mit $|w| \geq l \wedge \forall w = u_1 v u_2$ gilt $|v| \geq 1 \wedge |u_1 v| \leq l \implies u_1 v^* u_2 \not\subseteq L$*

Beweis. Sei $A = (Q, \Sigma, \delta, s, F)$ ein DFA mit $L(A) = L$. Setze dann $l = |Q|$

Sei $w \in L$ mit $|w| \geq l$, $w = a_1 \dots a_n$, wobei natürlich $n \geq l$. Dann ist $w \in L(A)$, also existiert $s = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$. Das sind mindestens $l + 1$ Zustände, also existieren $i < j \leq l$ mit $q_i = q_j$.



□

Beispiel 5.11. 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 ist $v \in a^+$, $u_1 a^0 u_2 = u_1 u_2 \notin L$

2. $L = \{u \in \{a, b\}^* \mid u \text{ Palindrom}\}$

zu $l \geq 1$ wähle $w = a^l b a^l$

Korollar 5.12 (Anwendungen des Satzes von Kleene)**.**

1. *reguläre Ausdrücke kann man komplementieren. Nach Kleene reicht es, entsprechendes für einen DFA zu zeigen:*

   Gegeben sei ein deterministischer endlicher Automat $A = (Q, \Sigma, \delta, s, F)$. Setze dann $\overline{A} = (Q, \Sigma, \delta, s, Q \setminus F)$, dann ist $L(\overline{A}) = \Sigma^* \setminus L(A)$.

(Achtung: für einen nichtdeterministischen Automat kann man diesen nicht so einfach komplementieren; → ϵ-Übergänge!)

2. *reguläre Ausdrücke kann man schneiden. (klar, denn* $r \cap s = \neg(\neg r + \neg s)$*).*

   Der Beweis kann aber auch über Automaten geführt werden, hier sogar über NFAs:

   Gegeben seien nichtdeterministische Automaten $A = (Q_A, \Sigma, \Delta_A, s_A, F_A)$ *und* $B = (Q_B, \Sigma, \Delta_B, s_B, F_B)$*.*

   Setze $A \times B := (Q_A \times Q_B, \Sigma, \Delta, (s_A, s_B), F_A \times F_B)$ *mit* $\Delta(a,(p,q)) = \underbrace{\Delta_A(a,p)}_{\subseteq Q_A} \times \underbrace{\Delta(a,q)}_{\subseteq Q_B} \subseteq Q_A \times Q_B$*.*

   oder anders gesagt aus

   - (p,q) *final* $\Leftrightarrow$ p *final und* q *final und*
   - $(p,q) \xrightarrow{a} (p',q') \Leftrightarrow p \xrightarrow{a} p'$ *und* $q \xrightarrow{a} q'$*.*

   folgt $L(A \times B) = L(A) \cap L(B)$*.*

5.3 Sprachen als Kodaten

Observe: Ein Automat A besteht aus einem *Transitionssystem* $A_0 = (Q, \Sigma, \delta, F)$ und einem Zustand $s \in A_0$. Verwechsle durch Currying δ mit $\delta : Q \to (\Sigma \to Q)$ und F mit $F : Q \to 2 := \{0,1\}$.

Dann „ist“ A_0:

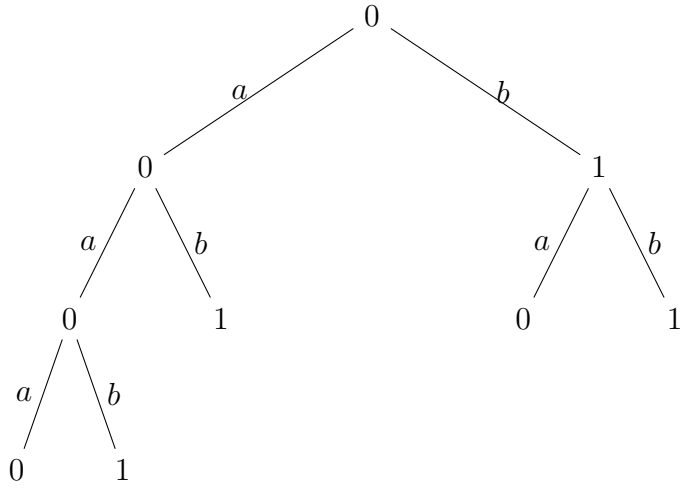
$$\langle F, \delta \rangle : Q \to 2 \times (\Sigma \to Q)$$

Wir stellen also fest, dass Transitionssysteme G-Koalgebren sind mit $GX = 2 \times (\Sigma \to X)$.

```
1 codata Lang Σ where
2         acc: Lang Σ -> Bool
3         der: Lang Σ -> (Σ -> Lang Σ)
```

beschreibt die finale G-Koalgebra mit Elementen:

- unendliche Bäume
- Knotenlabel in Bool
- je ein Kind pro $a \in \Sigma$



Knoten haben *Adressen* in Σ^*. Die unterste 1 hat beispielsweise die Adresse aab.

Der Baum ist vollständig beschrieben durch $\{u \in \Sigma^* \mid \text{Knoten } u \text{ hat Label1}\}$

Also sind diese Bäume isomorph zu Sprachen.

Struktur: $acc(L) = 1 \Leftrightarrow$ Wurzel des Baums L hat Label $1 \Leftrightarrow \epsilon \in L$ (denn ϵ ist Adresse der Wurzel). $der(L)(a) = a$-tes Kind der Wurzel des Baums $L = \{u \in \Sigma^* \mid au \in L\} =$ *Ableitung* von L nach $a =: L_a$

Menge Λ der Sprachen als Transitionssystem: L ist final $\Leftrightarrow \epsilon \in L :\Leftrightarrow L\downarrow$

$\delta(a, L) = L_a$

Transitionssystemmorphismen sind G-Koalgebra-Morphismen $A \to B$

$$\begin{array}{ccc} Q_A & \xrightarrow{\quad f \quad} & Q_B \\ {\scriptstyle \langle F_A, \delta_A \rangle}\downarrow & \# & \downarrow{\scriptstyle \langle F_B \delta_B \rangle} \\ 2 \times (\Sigma \to Q_A) & \xrightarrow[\underset{=\mathrm{id}_2}{\uparrow}]{2 \times (\Sigma \to f)} & 2 \times (\Sigma \to Q_B) \end{array}$$

1. Kommutieren in der linken Komponente:
   $F_B\Big(f(x)\Big) = F_A(x)$, d.h. x final $\Leftrightarrow f(x)$ final

2. Kommutieren in der rechten Komponente:

$$\Sigma \to f : (\Sigma \to Q_A) \to (\Sigma \to Q_B)$$
$$h \mapsto f \circ h$$

also: $\delta_B\Big(f(q)\Big)(a) = \Big(f \circ \delta_A(q)\Big)(a) = f\Big(\delta_A(q)(a)\Big)$

in alter Notation: $\delta_B\Big(a, f(q)\Big) = f\Big(\delta_A(a, q)\Big)$

Satz 5.13. *Zu* $A_0 = (Q, \Sigma, \delta_A, F_A)$ *ist*

$$f : Q \to \Lambda$$
$$q \mapsto L(\underbrace{A_0, q}_{Automat})$$

ein Transitionssystemmorphismus.

Beweis. q final $\Leftrightarrow \epsilon \in L(A_0, q) \Leftrightarrow L(A_0, q)$ final.

Ferner $f(\delta_A(a, q)) = \delta_A(a, f(q))$

$u \in f(\delta_{A_0}(a, q)) \Leftrightarrow \exists$Zustand $q' \in F_A$ mit $\delta_{A_0}(a, q) \xrightarrow{u} q' \Leftrightarrow \exists q' \in F_{A_0}$ mit $q \xrightarrow{au} q' \Leftrightarrow au \in L(A_0, q) \Leftrightarrow u \in L(A_0, q) = \delta(a, f(q))$

Also: der eindeutige Morphismus $A_0 \to \Lambda$ bildet jeden Zustand auf seine akzeptierte Sprache ab. □

Korollar 5.14. $L \in \Lambda$ *akzeptiert* L *(da* $\mathrm{id} : \Lambda \to \Lambda$ *Morphismus ist).*

5.4 Minimierung

Mit Λ bezeichnen wir die Menge der Sprachen mit einem finalen Transitionssystem: $Q \xrightarrow{\langle F, \delta \rangle} 2 \times (\Sigma \to Q)$

$\underset{\text{Finalitätsfunktion}}{\overset{\uparrow}{F}}(L) \Leftrightarrow \epsilon \in L,$

Definition 5.15. Die *Ableitung* einer Sprache L ist $\delta(L)(a) := L_a := \{w \in \Sigma^* \mid aw \in L\}$

Definition 5.16. $L(A, \cdot) : A \to \Lambda$ ist der eindeutige Morphismus, der einem Startzustand in dem Transitionssystem A die von dem dadurch definierten Automaten akzeptierte Sprache zuordnet.

Bemerkung 5.17 (Schreibweise)**.**
wir erweitern die Definition der δ-Zustandsübergangfunktion um $\delta(u, q) := q'$ für $q \xrightarrow{u} q'$

Definition 5.18. Für einen Zustand $q \in A$ aus einem Transitionssystem A setze das von q *erzeugte Untertransitionssystem*

$$\langle q \rangle := \{\delta(u, q) \mid u \in \Sigma^*\}$$

Lemma 5.19. *$\langle q \rangle$ ist abgeschlossen unter δ, also „ist“ $\langle q \rangle$ Transitionssystem.*

Lemma 5.20. *Offenbar ist $L(\langle q \rangle, q) = L(A, q)$*

letzte VL fixen, siehe tex-comment hier

$L(A, \cdot) : \langle q \rangle \overset{\text{surjektiv}}{\underset{\downarrow}{\twoheadrightarrow}} \langle L(A, q) \rangle$, denn $L(A, \delta(u, q)) = \delta(u, L(A, q)) \implies |\langle L(A, q) \rangle| \leq |\langle q \rangle| \leq |A|$ (Anzahl der Zustände)

Kurz: Wenn eine Sprache von einem Automaten (A, q) erzeugt wird, also $L(A, q) = L$, dann hat ist $|A| \geq |\langle L \rangle|$, das heisst, $\langle L \rangle$ ist der *minimale Automat* für L.

Korollar 5.21 (Satz von Myhill und Nerode)**.** *Eine Sprache L ist regulär $\Leftrightarrow$ $\langle L \rangle$ ist endlich.*

Zu $L \in \Lambda$ definiere eine Äquivaenzrelation $\sim_L$ auf Σ^* per

$$v \sim_L w :\Leftrightarrow \forall u \in \Sigma^* : (uv \in L \Leftrightarrow uw \in L)$$

Bemerkung 5.22 (Einschub)**.**
sei R Äquivalenzrelation auf einer Menge X. Dann bezeichnen wir die *Äquivalenzklasse* eines $x \in X$ mit $[x]_R := \{y \in X \mid x \,\mathrm{R}\, y\}$.

Es gilt nun $[x]_R \cup [y]_R = \emptyset$ für $\neg(x \,\mathrm{R}\, y)$.

Die *Quotientenmenge* ist definiert als $X/R := \{[x]_R \mid x \in X\}$

Satz 5.23 (Myhill und Nerode)**.** *Eine Sprache L ist regulär $\Leftrightarrow \Sigma^*/\sim_L$ ist endlich. („$\sim_L$ hat endlichen Index“).*

Beweis. Wir sind fertig, wenn $|\langle L \rangle| = |\Sigma^*/\sim_L|$.

Nun ist $L = \{L_v \mid v \in \Sigma^*\}$ und $\Sigma^*/\sim_L = \{[v]_{\sim_L} \mid v \in \Sigma^*\}$.

Damit folgt $L_v = L_w \Leftrightarrow \forall u.(u \in L_v \Leftrightarrow u \in L_w) \Leftrightarrow \forall u.(uv \in L \Leftrightarrow uw \in L) \Leftrightarrow v \sim_L w$ □

5.5 Reguläre Ausdrücke per Korekursion

Mache Transitionssystem $\mathcal{E} \to 2 \times (\Sigma \to \mathcal{E})$ aus der Menge $\mathcal{E}$ der regulären Ausdrücke.

```
1 data Expr Σ where
2         0,1: () -> Expr Σ
3         +, ·: Expr Σ -> Expr Σ -> Expr Σ
```

Basisfälle:

- $F(0) = \bot (= \text{False})$ und
  $\delta(0)(a) = 0$
- $F(1) = \top (= \text{True})$ und
  $\delta(1)(a) = 0$
- $F(a) = \bot (= \text{False})$ und
  $\delta(a)(a) = 1$, $\delta(a, b) = 0 \forall b \neq a$

Seien nun r und s reguläre Ausdrücke, definiere dann rekursiv:

- $F(r + s) = F(r) \vee F(s)$ und
  $\delta(r + s)(a) = \delta(r)(a) + \delta(s, a)$
- $F(rs) = F(r) \wedge F(s)$ und
  $$\delta(rs)(a) = \begin{cases} \delta(r)(a)s & \text{falls } F(r) = \bot (\Leftrightarrow L_a \ni \epsilon) \\ \delta(s)(a) + \delta(r)(a)s & \text{sonst} \end{cases}$$
- $F(r^*) = \top$ und
  $\delta(r^*)(a) = \delta(r)(a)r^*$

Das definiert $L(r)$ korekursiv per

- $\epsilon \in L(r) \Leftrightarrow F(r)$
- $L(r)_a = L(\delta(r)(a))$

Definition 5.24. Sei $A = (Q, \dots)$ ein Transitionssystem mit Zustandsmenge Q. Eine Relation $R \subseteq Q \times Q$ auf Q heißt *Bisimulation*, wenn gilt:

- $\forall p \,\mathrm{R}\, q : \; p \in F \Leftrightarrow q \in F$
- $\delta(a, p) \,\mathrm{R}\, \delta(a, q)$ für alle $a \in \Sigma$

Lemma 5.25. *Sei $f : A \to \Lambda$ Morphismus, R Bisimulation auf A, und es gelte $p \,\mathrm{R}\, q$. Dann gilt $f(p) = f(q)$, d.h. p und q akzeptieren dieselbe Sprache.*

Beweis. Definiere $f^2[R] := \Big\{\Big(f(p'), f(q')\Big) \mid (p', q') \in R\Big\}$ ist Bisimulation, da f Morphismus ist. □

Beispiel 5.26 (Gleichheit (der erzeugten Sprachen) von regulären Ausdrücken)**.**

1. zu zeigen: $r + 0 = r$

   Behauptung: $R = \{(r + 0, r) \mid r \in \mathcal{E}\}$ ist Bisimulation, denn

   - $F(r + 0) = F(r) \vee F(0) = F(r) \vee \bot = F(r)$
   - $\delta(r + 0)(a) = \delta(r)(a) = \delta(0)(a) = \delta(r)(a) + 0 \,\mathrm{R}\, \delta(r)(a)$ ✓

2. zu zeigen: $r(s + t) = rs + rt$

   Behauptung: $R = \Big\{\Big(r(s + t), rs + rt\Big) \mid r, s, t \in \mathcal{E}\Big\}$ ist Bisimulation, denn

   - $F\Big(r\ (s + t)\Big) = F(r) \wedge \Big(F(s) \vee F(t)\Big) = \Big(F(r) \wedge F(s)\Big) \vee \Big(F(r) \wedge F(t)\Big) = F(rs + rt)$
   - $\delta\Big(r\ (s + t)\Big)(a) = \delta(s)(a) + \delta(t)(a) + \delta(r)(a)\ (s + t)\ \underset{\substack{\uparrow \\ \text{Kontextabschluss von } R}}{C(R)}\ \delta(s)(a) + \delta(t)(a) + \delta(r)(a)\ s + \delta(r)(a)\ t = \delta(rs)(a) + \delta(rt)(a)$

   → Bisimulation „up-to“ (hier: Kontext, Bisimulation)