Theorie der Programmierung

SoSe 2023

Dieses Skript zur Vorlesung *„Theorie der Programmierung“* (Prof. Dr. Lutz Schröder) wurde im Sommersemester 2014 von den untenstehenden Studenten erarbeitet und vom Veranstalter ab Sommersemester 2015 überarbeitet.

Florian Jung `florian.jung@fau.de`

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

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Was tut ein Programm?

- Terminiert es, bzw. bleibt es (bei reaktiven Programmen) nicht stecken?
- Liefert es korrekte Ergebnisse, bzw. verhält es sich richtig?

Plan:

- Termersetzung
- λ-Kalkül (LISP)
- (Ko-)Datentypen, (Ko-)Induktion
- reguläre Ausdrücke und Minimierung von Automaten

1.1 Literatur

Termersetzungssysteme (TES):

- F. Baader & T. Nipkow: Term Rewriting and all that, Cambridge University Press, 1998.
- J. W. Klop: Term rewriting systems, in S. Abramsky, D. Gabbay and T. Maibaum (eds.), Handbook of Logic in Computer Science, Oxford University Press, 1992.
- J. Giesl: Termersetzungssysteme, Vorlesungsskript SoSe 2011, RWTH Aachen.

λ-Kalkül:

- H. Barendregt: Lambda Calculi with types, in S. Abramsky, D. Gabbay and T. Maibaum (eds.), Handbook of Logic in Computer Science Vol. II, Oxford University Press, 1992.
- T. Nipkow: Lambda-Kalkül, Vorlesungskript TU München, 2004.
- Masako Takahashi. Parallel reductions in λ-calculus.

(Ko-)Induktion

- B. Jacobs and J. Rutten: A Tutorial on (Co-)Algebras and (Co-)Induction, EATCS Bulletin 42 (1997), 222–259.
- J. Rutten: Automata and Coinduction – an Exercise in Coalgebra, Proc. CONCUR 1998, Lect. Notes Comput. Sci. 1466, 194–218, Springer, 2006.

Reguläre Ausdrücke und endliche Automaten:

- J. Hopcroft, J. D. Ullmann and R. Motwani: Introduction to Automata Theory, Formal Languages and Computation, 3rd ed., Prentice Hall, 2006.

1.2 Konventionen

Natürliche Zahlen $0 \in \mathbb{N}$

Logische Implikation Für den Folge- und Äquivalenzpfeil werden, anders als in GLoIn, die Symbole „⇒“ sowie „⇔“ genutzt. Das Symbol „→“ wird anderweitig benötigt.

2 Termersetzungssysteme (TES)

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

Anwendungen:

- (Algebraische) Spezifikation
- Programmverifikation
- automatisches Beweisen
- Computeralgebra (Gröbnerbasen / Buchbergeralgorithmus)
- Programmierung:
  - Termersetzung ist Turing-vollständiger Formalismus
  - Grundlage funktionaler Programmiersprachen

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

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

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

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

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

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

Beweis:

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

Beispiel 2.4 (Optimierung)**.**

$$plus\ (plus\ x\ y)\ z = plus\ x\ (plus\ y\ z)$$

Bei Auswertung der linken Seite der Gleichung gemäß der Definition von *plus* werden weniger Schritte benötigt als für die rechte Seite (wie viele jeweils?).

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

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

umformen. Bei der Auswertung z.B. des Terms

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

bekommen wir dann (nicht unbedingt unlösbare) Probleme mit der Terminierung, da die Umformung bei naiver Herangehensweise in eine Schleife läuft.

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

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

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

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

und

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

2.1 Syntax und operationale Semantik

Wir führen nun die grundlegenden Definitionen zu Termersetzungssystemen ein. Diese betreffen, wie letztlich bei jedem Programmierformalismus, zum einen die syntaktische Präsentation von Termersetzungssystemen und zu anderen ihr operationales Verhalten. Letzteres ist durch Reihe von durch ein Termersetzungssystem induzierten binären Relationen auf Termen gegeben. Wir erinnern zunächst an eine Reihe von Begriffen und Konstruktionen im Bereich binäre Relationen und Terme.

2.1.1 Recall: Binäre Relationen

Definition 2.7. Eine *binäre Relation* zwischen zwei Mengen X und Y ist eine Teilmenge $R \subseteq X \times Y$. Wir schreiben oft xRy für $(x, y) \in R$.

Beispiel 2.8. Die Kleiner-Gleich-Relation auf den natürlichen Zahlen ist eine Teilmenge $\leqslant\ \subseteq \mathbb{N} \times \mathbb{N}$, nämlich die Teilmenge $\leqslant\ = \{(n, m) \mid \exists k \in \mathbb{N}.\, n + k = m\}$.

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

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

Beispiel 2.10. Die Relation $\leqslant$ auf natürlichen Zahlen ist eine Präordnung (sogar eine *partielle Ordnung*, d.h. zusätzlich noch *antisymmetrisch* – wenn für $n, m \in \mathbb{N}$ sowohl $n \leqslant m$ als auch $m \leqslant n$ gilt, so folgt $n = m$). Dagegen ist $\leqslant$ *keine* Äquivalenzrelation, da $\leqslant$ klarerweise nicht symmetrisch ist. Eine Äquivalenzrelation auf den natürlichen Zahlen ist z.B. Kongruenz modulo $k \in \mathbb{N}$: Zahlen $n, m \in \mathbb{N}$ sind *kongruent modulo* k, in Symbolen $n \equiv_k m$, wenn $n - m$ durch k teilbar ist. Man prüft leicht nach, dass $\equiv_k$ reflexiv, symmetrisch und transitiv ist (letzteres verwendet, dass die Summe von durch k teilbaren Zahlen wieder durch k teilbar ist).

Definition 2.11 (Standardkonstruktionen auf Relationen)**.**

- *Gleichheit* („=“): Die *Identitätsrelation id* (oder *Diagonale* Δ) auf einer Menge X ist gegeben durch $id = \Delta = \{(x, x) \mid x \in X\}$. Diese Relation ist offenbar eine Äquivalenz, die kleinste Äquivalenz auf X.
- Die *Verkettung* oder *Komposition* $R \circ S \subseteq X \times Z$ von Relationen $R \subseteq Y \times Z$ und $S \subseteq X \times Y$ ist definiert als

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

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

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

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

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

Beispiel 2.12. Die Umkehrrelation $\leqslant^-$ von $\leqslant$ ist $\geqslant$. Die Verkettung von $\leqslant$ mit sich selbst ist wieder $\leqslant$. Dagegen ist die Verkettung $< \circ <$ der Relation $<$ auf natürlichen Zahlen mit sich selbst die Relation

$$\{(n, m) \in \mathbb{N} \mid n + 2 \leqslant m\}.$$

Auf rationalen Zahlen wiederum ist die Verkettung von $<$ mit sich selbst wieder $<$ (warum?).

Lemma 2.13. *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 2.14. Sei $R \subseteq X \times X$ eine Relation. Der *reflexive/symmetrische/transitive Abschluss* von R ist die kleinste reflexive/symmetrische/transitive Relation, die R enthält.

Wir haben folgende explizite Darstellungen der verschiedenen Abschlüsse von R:

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

$$\begin{aligned} R^+ &:= R \ \cup \ (R \circ R) \ \cup \ (R \circ R \circ R) \ \cup \ \dots \\ &= \bigcup_{n=1}^{\infty} R^n \\ &(= \{(x, y) \mid \exists n \geqslant 1.\, (x, y) \in R^n\}), \end{aligned}$$

  also

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

  (man beachte, dass die von den x_i gebildete R-Kette n R-Schritte mit $n \geqslant 1$ enthält, also mindestens einen).

- Transitiv-reflexiver Abschluss:

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

  also

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

  (hier haben wir $n \geqslant 0$ R-Schritte, also möglicherweise keinen).

Lemma 2.15 (Erzeugte Äquivalenz)**.** *Seien $R, S \subseteq X \times X$.*

1. *Wenn S symmetrisch ist, dann sind auch S^+ und $S^\star$ symmetrisch.*

2. $(R \cup R^-)^\star$ *ist symmetrisch.*
3. $(R \cup R^-)^\star$ *ist die von* R *erzeugte Äquivalenz (d.h. die kleinste Äquivalenz, die* R *enthält).*

Beweis. Zu 1.: Wenn x über einen Pfad $x = x_0 S x_1 \dots x_{n-1} S x_n = y$ mit y in Relation S^+ oder $S^\star$ steht, dann kann man per Symmetrie diesen Pfad auch in der entgegengesetzten Richtung ablaufen, d.h. man hat $y = x_n S x_{n+1} \dots x_1 S x_0 = x$.

Es folgt sofort 2., und dass $(R \cup R^-)^\star$ in 3. eine Äquivalenz ist. Zu zeigen ist dann noch, dass dies die kleinste Äquivalenz ist, die R enthält. Dies folgt sofort über die oben gegebenen Beschreibungen des symmetrischen und des transitiv-reflexiven Abschlusses: Sei S eine Äquivalenzrelation mit $R \subseteq S$. Da $R \cup R^-$ der symmetrische Abschluss von R ist und S insbesondere symmetrisch ist, folgt $R \cup R^- \subseteq S$. Da $(R \cup R^-)^*$ der reflexiv-transitive Abschluss von $R \cup R^-$ ist und S insbesondere reflexiv und transitiv ist, folgt dann $(R \cup R^-)^* \subseteq S$. □

Beispiel 2.16. Sei $R \subseteq \mathbb{N} \times \mathbb{N}$ die Relation, die jede positive natürliche Zahl mit ihrem unmittelbaren Vorgänger in Beziehung setzt:

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

Dann

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

2.1.2 Recall: Terme

Wir verwenden im wesentlichen denselben Begriff von Signatur und Term wie in GLoIn, benötigen hier aber keine Prädikatensymbole, sondern nur Funktionssymbole. Im einzelnen: Die Signatur legt fest, welche Funktionssymbole in Termen verwendet werden dürfen, und Terme sind definiert als aus Variablen und Funktionssymbolen unter Einhaltung der Stelligkeit von Funktionssymbolen zusammengesetzt. Formal:

Definition 2.17.

- Eine *Signatur* Σ ist eine Menge von *Funktionssymbolen* $f, g, \dots$ jeweils gegebener Stelligkeit. Wir schreiben $f/n \in \Sigma$, wenn f ein n-stelliges Funktionssymbol in Σ ist. Eine *Konstante* ist ein nullstelliges Funktionssymbol.
- Sei V eine Menge von *Variablen*. Ein *Term über* V ist dann induktiv definiert wie folgt:

  - Sei $x \in V$. Dann ist x, also eine einzelne Variable, ein Term.
  - Seien $t_1, \ldots, t_n$ Terme und sei $f/n \in \Sigma$. Dann ist $f(t_1, \ldots, t_n)$ ein Term. (Damit ist insbesondere jede Konstante ein Term.)

  Induktiv bedeutet hierbei, dass ein Objekt dann ein Term ist, wenn sich dies durch endlich viele Anwendungen dieser beiden Regeln herleiten lässt. Mit anderen Worten sind Terme t über V definiert durch die Grammatik

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

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

Wir benötigen eine formale Definition der in einem Term vorkommenden Variablen. Wir nennen solche Variablen bereits jetzt *frei*, obwohl zur Zeit noch keine Konstrukte zur Variablenbindung vorkommen; diese werden aber in späteren Erweiterungen hinzukommen.

Definition 2.18 (Freie Variablen)**.** Die Menge $FV(t)$ der *freien Variablen in* t ist rekursiv definiert durch

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

Für Substitutionen verwenden wir eine von der in GLoIn gewählten formal leicht abweichende (aber letztlich äquivalente) Repräsentation:

Definition 2.19 (Substitution)**.** Eine *Substitution* ist eine Abbildung $\sigma : V_0 \to T_\Sigma(V)$ für eine endliche Teilmenge $V_0 \subseteq V$, d.h. eine Vorschrift zur Ersetzung von Variablen durch Terme. Wenn nichts anderes gesagt ist, nehmen wir typischerweise an, dass V_0 nur „relevante“, d.h. in den Termen, auf die wir σ anwenden, tatsächlich vorkommende Variablen enthält. Wir schreiben

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

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

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

$$x\sigma = \begin{cases} \sigma(x) & \text{falls } x \in V_0 \\ x & \text{sonst} \end{cases}$$
$$f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$$

Beispiel 2.20. Wir wählen $\Sigma = \{+/2, */2\}$ und notieren $+, *$ wie üblich in Infixschreibweise. Dann haben wir

$$(x + y)[x * y/x, y * y/y] = x * y + y * y.$$

Dies illustriert insbesondere, dass die Substitution $[x*y/x, y*y/y]$ *simultan* ist, d.h. gleichzeitig x durch $x * y$ und y durch $y * y$ ersetzt (was wäre $(x + y)[x * y/x][y * y/y]$?).

Definition 2.21 (Kontext)**.**

1. Ein *Kontext* ist ein Term $C(\cdot)$ mit genau einer Freistelle $(\cdot)$. Formal sind Kontexte definiert durch die Grammatik

$$C(\cdot) ::= (\cdot) \mid f(t_1, \ldots, t_{i-1}, C(\cdot), t_{i+1}, \ldots, t_n) \qquad (f/n \in \Sigma, 1 \leqslant i \leqslant n),$$

   die wie beabsichtigt sicherstellt, dass $(\cdot)$ in $C(\cdot)$ genau einmal vorkommt.

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

$$(\cdot)(t) = t$$
$$f(t_1, \ldots, C(\cdot), \ldots, t_n)(t) = f(t_1, \ldots, C(t), \ldots, t_n)$$

2.1.3 Termersetzungssysteme: Formale Definition

Definition 2.22.

1. Ein *Termersetzungssystem* ist eine Relation

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

   die Elemente von $\to_0$ nennen wir *Ersetzungsregeln.*

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

   - *abgeschlossen* bezüglich eines Kontexts $C(\cdot)$, wenn

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

     für alle Terme s, t;
   - *kontextabgeschlossen*, wenn R abgeschlossen bezüglich aller Kontexte ist;
   - *stabil*, wenn

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

     für jede Substitution σ und alle Terme t, s.

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

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

4. Die *Reduktionsrelation* oder einfach *Reduktion* ist der transitiv-reflexive Abschluss $\to^\star$ von $\to$. Ein Term t *reduziert* auf einen Term s, wenn $t \to^\star s$.

5. *Konvertierbarkeit* $\leftrightarrow^\star = (\to \cup \to^-)^\star$ ist die von $\to$ erzeugte Äquivalenz.

6. Ein Term t heißt *normal*, wenn t nicht reduziert werden kann, d.h. wenn kein Term s mit $t \to s$ existiert. Wir schreiben dann $t \not\to$.

7. Ein Term s heißt eine *Normalform* eines Terms t, wenn s normal ist und $t \to^\star s$.

Die Überprüfung von Kontextabgeschlossenheit lässt sich etwas vereinfachen, d.h. auf bestimmte Kontexte beschränken:

Lemma 2.23. *Sei $R \subseteq T_\Sigma(v) \times T_\Sigma(V)$.*

1. *R ist bereits dann kontextabgeschlossen, wenn R abgeschlossen bezüglich aller Kontexte der Form $f(t_1, \ldots, t_{i-1}, (\cdot), t_{i+1}, \ldots, t_n)$ (für $f/n \in \Sigma$ und Terme t_i) ist.*
2. *Wenn R stabil ist, ist R ist bereits dann kontextabgeschlossen, wenn R abgeschlossen bezüglich aller Kontexte der Form $f(x_1, \ldots, x_{i-1}, (\cdot), x_{i+1}, \ldots, x_n)$ für Variablen x_i ist.*

Beweis. 1. Strukturelle Induktion über Kontexte $C(\cdot)$: Der Basisfall $(\cdot)$ ist trivial. Für einen Kontext der Form $f(t_1, \ldots, C(\cdot), \ldots, t_n)$ rechnen wir wie folgt:

$$\begin{aligned} tRs &\implies C(t)RC(s) && \text{(IV)} \\ &\implies f(t_1, \ldots, C(t), \ldots, t_n)Rf(t_1, \ldots, C(s), \ldots, t_n) && \text{(Annahme).} \end{aligned}$$

2. Wir verwenden Teil 1, d.h. es reicht Abgeschlossenheit bezüglich $f(t_1, \ldots, t_{i-1}, (\cdot), t_{i+1}, \ldots, t_n)$: Für frische Variablen x_i haben wir

$$\begin{aligned} tRs &\implies f(x_1, \ldots, x_{i-1}, t, x_{i+1}, \ldots, x_n)Rf(x_1, \ldots, x_{i-1}, s, x_{i+1}, \ldots, x_n) && \text{(Annahme)} \\ &\implies f(t_1, \ldots, C(t), \ldots, t_n)Rf(t_1, \ldots, C(s), \ldots, t_n) && \text{(Stabilität).} \end{aligned}$$

□

Beispiel 2.24. Wir definieren $\to_0$ durch

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

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

Wenn wir jetzt unterstellen, dass Σ außerdem Konstanten a, b, c, d, e enthält, dann haben wir

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

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

Beispiel 2.25. Sei $\Sigma = \{+/2, s/1, 0/0\}$, und sei $\to_0$ definiert durch

$$s(x) + y \to_0 s(x + y) \tag{1}$$

$$0 + y \to_0 y \tag{2}$$

$$(x + y) + z \to_0 x + (y + z) \tag{3}$$

Wir formen nun den Ausdruck $(s(x) + s(y)) + z$ um. Wir stellen fest, dass wir zwei verschiedene Regeln anwenden können; dies ist im Detail in Abbildung 1 dargestellt. Der letzte Term $s(x) + s(y + z)$ ist eine Normalform. Wir sehen, dass es *in diesem Fall* keine Rolle spielt, welche Wahl wir am Anfang treffen; beide Reduktionen führen hier auf dieselbe Normalform.

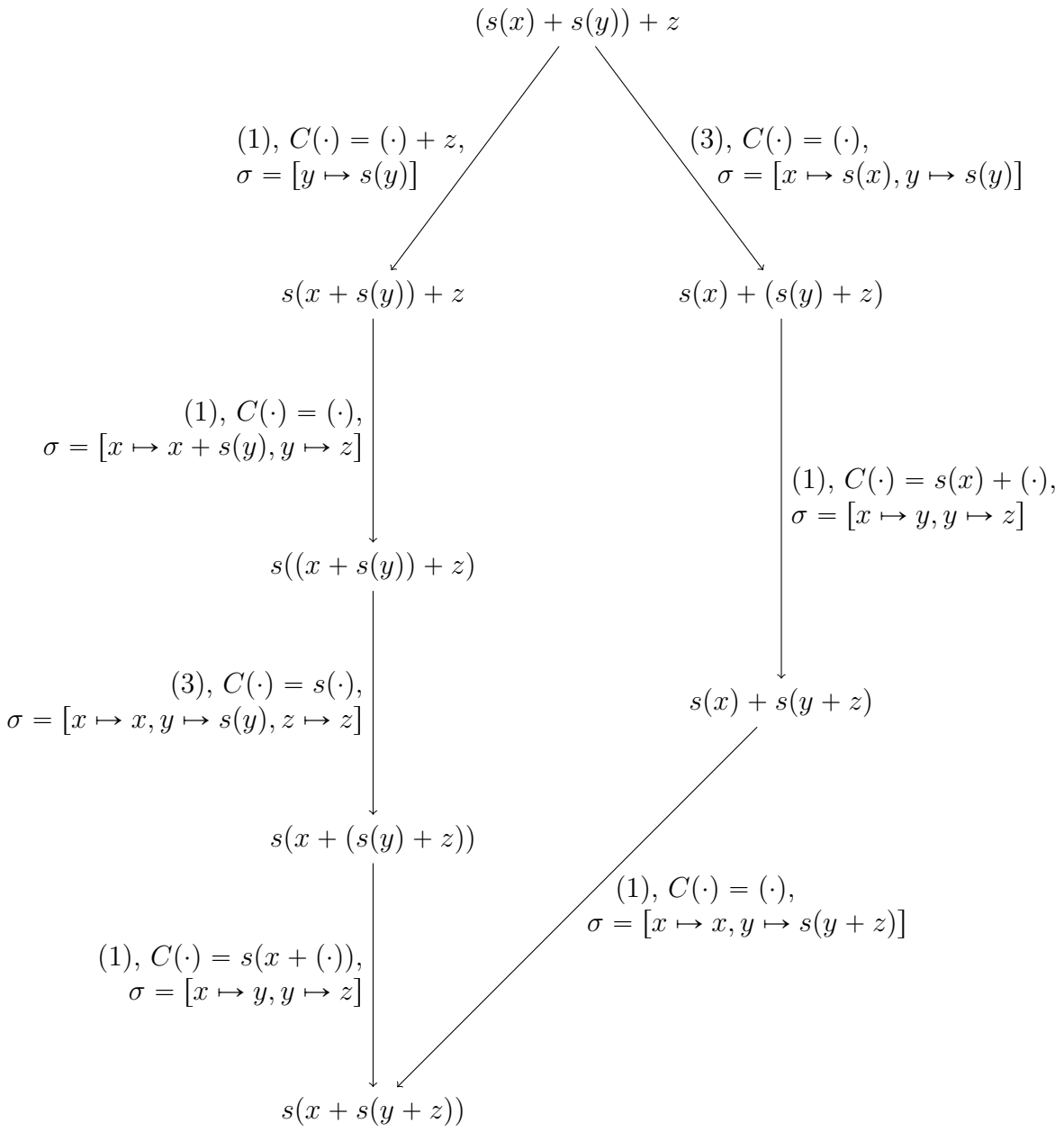


Abbildung 1: Zwei verschiedene Reduktionswege

2.2 Terminierung

Wir wenden uns nunmehr der Frage zu, wann die Termreduktion terminiert. Die formalen Definition zu diesem Begriff gestalten sich wie folgt.

Definition 2.26. Eine Relation $R \subseteq X \times X$ heißt *wohlfundiert* (well-founded, wf.), wenn es *keine* unendliche Folge $(x_i)_{i\in\mathbb{N}}$ gibt mit $x_0 R x_1 R x_2 \ldots$.

Beispiel 2.27. $(\mathbb{N}, >)$ ist wohlfundiert, $(\mathbb{Z}, >)$ nicht, $(\mathbb{R}_{\geqslant 0}, >)$ auch nicht.

Definition 2.28. Ein Term t heißt

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

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

Beispiel 2.29. Wir definieren $\to_0$ durch

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

- $g(x)$ ist stark normalisierend: $g(x) \to 1 \not\to$, und $g(x)$ hat keine weiteren Reduktionen.
- $f(3)$ ist nicht schwach normalisierend: $f(3) \to f(3) \to f(3) \to \ldots$, und $f(3)$ hat keine weiteren Reduktionen.
- $g(f(3))$ ist schwach normalisierend: $g(f(3)) \to 1 \not\to$
- $g(f(3))$ ist nicht stark normalisierend: $g(f(3)) \to g(f(3)) \to \ldots$

2.2.1 Reduktionsordnungen

Nach der Klärung des *Begriffs* der Terminierung stellt sich nun die Frage, wie man Terminierung in konkreten Fällen *beweist*. Wir gehen hierzu in zwei Stufen vor: Wir führen zunächst auf relativ abstrakter Ebene sogenannte Reduktionsordnungen ein, deren rein axiomatisch definierte Eigenschaften Terminierung garantieren, wenn alle Reduktionen Terme unter der Reduktionsordnung verkleinern. Im nächsten Abschnitt behandeln wir dann eine konkrete Methode, Reduktionsordnungen mittels polynomieller Interpretationen zu konstruieren.

Definition 2.30.

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. Eine stabile, kontextabgeschlossene und wohlfundierte strikte Ordnung $R \subseteq T_\Sigma(v) \times T_\Sigma(V)$ heißt eine *Reduktionsordnung*.

Satz 2.31. *Sei $>$ Reduktionsordnung, und für alle Terme t, s gelte: Aus $t \to_0 s$ folgt $t > s$. Dann ist $\to$ stark normalisierend.*

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

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

Beispiel 2.32. Sei $|t|$ die Größe von t. Betrachte die durch $t > s :\Leftrightarrow |t| > |s|$ definierte Relation $>$.

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

Das Problem im Gegenbeispiel für Stabilität ist, dass y rechts häufiger vorkommt als links. Wenn das allerdings nicht vorkommt, d.h. wenn für $t \to_0 s$ jede Variable in s höchstens so oft vorkommt wie in t, dann ist $>$ offensichtlich stabil; wenn dann außerdem $t \to_0 s$ stets $|t| > |s|$ impliziert, ist $\to$ somit SN.

Beispiel 2.33. Definiere $>$ durch $t > s :\Leftrightarrow s$ ist echter Unterterm von t. Damit ist $>$ wohlfundiert und stabil, aber nicht kontextabgeschlossen: Es gilt $x + x > x$, aber nicht $f(x + x) > f(x)$.

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

Beispiel 2.34. $\emptyset$ ist eine Reduktionsordnung.

Beispiel 2.35. $\to^+$ ist eine Reduktionsordnung, wenn $\to$ stark normalisierend ist.

2.2.2 Polynomordnungen

Wie angekündigt geben wir jetzt eine systematische Methode zur Konstruktion spezieller Reduktionsordnungen, sogenanter Polynomordnungen, an.

Recall: Polynome Die Menge der Polynome *über* $\mathbb{N}$, d.h. mit natürlichzahligen Koeffizienten, in Variablen $x_1, \ldots, x_n$ schreiben wir

$$\mathbb{N}[x_1, \ldots, x_n] = \left\{ \textstyle\sum_{i_1,\ldots,i_n \in \mathbb{N}} a_{i_1,\ldots,i_n} x_1^{i_1} \ldots x_n^{i_n} \mid a_{i_1,\ldots,i_n} \in \mathbb{N}, a_{i_1,\ldots,i_n} = 0 \text{ fast immer} \right\};$$

z.B. $x^2y + 2y^2zx \in \mathbb{N}[x, y, z]$. Formal ist ein Polynom zunächst ein syntaktisches Objekt, d.h. eine formale Summe, die einfach durch die Wahl der Koeffizienten (d.h. der $a_{i_1,\ldots,i_n}$ in obiger Darstellung) gegeben ist. Ein Polynom $p \in \mathbb{N}[x_1, \ldots, x_n]$ *definiert* die Funktion

$$\mathbb{N}^n \to \mathbb{N} \qquad (k_1, \ldots, k_n) \mapsto p(k_1, \ldots, k_n) \in \mathbb{N},$$

die p für gegebene Werte aus $\mathbb{N}$ auswertet; durch diese Funktion ist die *Semantik* von p gegeben. Wir werden die Unterscheidung zwischen dem syntaktischen Objekt p und seiner Semantik hier aber nicht überbetonen.

Summen und Produkte von Polynomen sind wieder Polynome (nach Zusammenfassen bzw. Ausmultiplizieren und Zusammenfassen). Deswegen ergibt das Einsetzen von Polynomen in ein Polynom wieder ein Polynom. Seien $p \in \mathbb{N}[x_1, \ldots, x_n]$ und $q_1, \ldots, q_n \in \mathbb{N}[y_1, \ldots, y_m]$; dann schreiben wir

$$p(q_1, \ldots, q_n) \in \mathbb{N}[y_1, \ldots, y_m]$$

für das Polynom, das aus p durch Einsetzen der Polynome q_i für die Variablen $x_1, \ldots, x_n$ entsteht.

Definition 2.36. Für $\emptyset \neq A \subseteq \mathbb{N}\backslash\{0\}$ definieren wir eine Ordnung $>_A$ auf $\mathbb{N}[x_1, \ldots, x_n]$ durch

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

Beispiel 2.37. $x^2 >_{\mathbb{N}_{\geqslant 2}} x$ (mit $\mathbb{N}_{\geqslant 2} = \{n \in \mathbb{N} \mid n \geqslant 2\}$).

Lemma 2.38. *$>_A$ ist wohlfundiert.*

Beweis. Wähle $a \in A$. Man nehme an, es gäbe eine Folge $(p_i)_{i\in\mathbb{N}}$ mit $p_0 >_A p_1 >_A \ldots$. Dann gälte $\underbrace{p_0(a, \ldots, a)}_{\in\mathbb{N}} > p_1(a, \ldots, a) > \ldots$, im Widerspruch zur Wohlfundiertheit von $\mathbb{N}$. □

Definition 2.39. Ein Polynom $p = \sum a_{i_1,\ldots,i_n} x_1^{i_1} \ldots x_n^{i_n} \in \mathbb{N}[x_1, \ldots, x_n]$ heißt *streng monoton*, wenn jedes x_j in p vorkommt, formal: $\forall j \in \{1, \ldots, n\}.\, \exists i_1, \ldots, i_n \geqslant 0.\, i_j > 0 \wedge a_{i_1,\ldots,i_n} > 0$.

Lemma 2.40. *Wenn p streng monoton im Sinne von Definition 2.39 ist, dann ist die zu p gehörige Funktion auf $\mathbb{N}\backslash\{0\}$ streng monoton im üblichen Sinn, d.h. wenn $k_1, \ldots, k_n, l_1, \ldots, l_n \in \mathbb{N}\backslash\{0\}$ mit $k_j \geqslant l_j$ für alle j und $k_j > l_j$ für mindestens ein j, dann gilt $p(k_1, \ldots, k_n) > p(l_1, \ldots, l_n)$.*

Beweis. Für jeden Term $a_{i_1,\ldots,i_n} x_1^{i_1} \ldots x_n^{i_n}$ in p gilt

$$a_{i_1,\ldots,i_n} k_1^{i_1} \ldots k_n^{i_n} \geqslant a_{i_1,\ldots,i_n} l_1^{i_1} \ldots l_n^{i_n}, \qquad (+)$$

da $a_{i_1,\ldots,i_n} \geqslant 0$. Ferner gibt es in p einen Term $a_{i_1,\ldots,i_n} x_1^{i_1} \ldots x_n^{i_n}$ mit $a_{i_1,\ldots,i_n} > 0$ und $i_j > 0$, und für diesen Fall gilt in $(+)$ sogar $>$, da die k_i nach Annahme positiv sind; also gilt insgesamt $>$. □

Definition 2.41. Eine *(monotone) polynomielle Interpretation* $\mathcal{A}$ für Σ besteht aus

- einem streng monotonen Polynom $0 \neq p_f \in \mathbb{N}[x_1, \ldots, x_n]$ für jedes $f/n \in \Sigma$
- einer Teilmenge $\emptyset \neq A \subseteq \mathbb{N}\backslash\{0\}$, die unter den p_f *abgeschlossen* ist, d.h. $p_f(a_1, \ldots, a_n) \in A$ für alle $a_1, \ldots, a_n \in A$ und alle $f/n \in \Sigma$.

(In typischen Beispielen ist A von der Form $A = \mathbb{N}_{\geqslant k} = \{n \in \mathbb{N} \mid n \geqslant k\}$ für ein $k \in \mathbb{N}$; in diesem Fall gilt Abgeschlossenheit automatisch.) Die hierdurch induzierte *Polynomordnung* auf Termen ist definiert durch

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

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

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

Lemma 2.42 (Substitutionslemma für polynomielle Interpretationen)**.** *Sei* $t \in T_\Sigma(\{x_1, \ldots, x_n\})$ *und* $\sigma = [t_1/x_1, \ldots, t_n/x_n]$. *Dann gilt*

$$p_{t\sigma} = p_t(p_{t_1}, \ldots, p_{t_n}).$$

Beweis. Induktion über t.

Induktionsanfang:

$$p_{x_i\sigma} = p_{t_i} = x_i(p_{t_1}, \ldots, p_{t_n}) = p_{x_i}(p_{t_1}, \ldots, p_{t_n})$$

Im Induktionsschritt haben wir

$$\begin{aligned}
p_{f(s_1,\ldots,s_k)\sigma} &= p_{f(s_1\sigma,\ldots,s_n\sigma)} \\
&= p_f(p_{s_1\sigma}, \ldots, p_{s_k\sigma}) && \text{(Definition)} \\
&= p_f(p_{s_1}(p_{t_1}, \ldots, p_{t_n}), \ldots, p_{s_k}(p_{t_1}, \ldots, p_{t_n})) && \text{(IV)} \\
&= p_f(p_{s_1}, \ldots, p_{s_k})(p_{t_1}, \ldots, p_{t_n}) && \text{(Substitution)} \\
&= p_{f(s_1,\ldots,s_n)}(p_{t_1}, \ldots, p_{t_n}) && \text{(Definition).}
\end{aligned}$$

Im mit „Substitution“ markierten Schritt nutzen wir dabei aus, dass das Einsetzen in Polynome sich im Wesentlichen wie Substitution verhält, nur, dass dabei eben noch modulo Gleichheit von Polynomen gerechnet wird (also modulo Distributivgesetz etc.); analog gilt eben für jeden Term t und Substitutionen σ, τ, dass $(t\sigma)\tau = t(\sigma\tau)$. □

Satz 2.43. *Polynomordnungen sind Reduktionsordnungen.*

Beweis. Sei $\mathcal{A}$ eine polynomielle Interpretation; der Einfachheit halber schreiben wir nur $\succ$ für die durch $\mathcal{A}$ induzierte Polynomordnung.

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

- $\succ$ ist stabil: Sei $t \succ s$, d.h. $p_t >_A p_s$, und sei $\sigma = [t_1/x_1, \ldots, t_n/x_n]$. Zu zeigen ist $t\sigma \succ s\sigma$, d.h. $p_{t\sigma} >_A p_{s\sigma}$. Seien also $k_1, \ldots, k_m \in A$, wobei m die Anzahl der freien Variablen in $t_1, \ldots, t_n$ ist. Per Abgeschlossenheit von A gilt $p_{t_i}(k_1, \ldots, k_m) \in A$ für $i = 1, \ldots, n$, also

$$\begin{aligned} p_{t\sigma}(k_1, \ldots, k_m) &= p_t(p_{t_1}(k_1, \ldots, k_m), \ldots, p_{t_n}(k_1, \ldots, k_m)) && \text{(Lemma 2.42)} \\ &> p_s(p_{t_1}(k_1, \ldots, k_m), \ldots, p_{t_n}(k_1, \ldots, k_m)) && (p_t >_A p_s) \\ &= p_{s\sigma}(k_1, \ldots, k_m) && \text{(Lemma 2.42).} \end{aligned}$$

- $\succ$ ist kontextabgeschlossen: Da $\succ$ stabil ist, reicht es nach Lemma 2.23, Abschluss unter Kontexten der Form $f(x_1, \ldots, x_{i-1}, (\cdot), x_{i+1}, \ldots, x_n)$ zu zeigen. Sei also $t \succ s$, d.h. $p_t >_A p_s$; zu zeigen ist $f(x_1, \ldots, x_{i-1}, t, x_{i+1}, \ldots, x_n) \succ f(x_1, \ldots, x_{i-1}, s, x_{i+1}, \ldots, x_n)$, also

$$p_f(k_1, \ldots, p_t(k_1, \ldots, k_n), \ldots, k_n) > p_f(k_1, \ldots, p_s(k_1, \ldots, k_n), \ldots, k_n) \qquad (*)$$

  für $k_1, \ldots, k_n \in A$. Da $p_t >_A p_s$, gilt $p_t(k_1, \ldots, k_n) > p_s(k_1, \ldots, k_n)$; $(*)$ folgt dann wegen $A \subseteq \mathbb{N}\backslash\{0\}$ per Lemma 2.40, da p_f streng monoton ist.

□

Korollar 2.44. *Sei $\to_0$ ein Termersetzungssystem und $>_{\mathcal{A}}$ eine Polynomordnung. Falls*

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

dann ist $\to$ stark normalisierend.

Beweis. Unmittelbar aus Satz 2.43 und Satz 2.31. □

Beispiel 2.45. Wir definieren ein TES $\to_0$ durch

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

(Achtung: Das ist a priori nur für Zwecke der schwachen Normalisierung äquivalent zum System mit nur einer Ersetzungsregel $(x \oplus y) \oplus z \to_0 y \oplus y$; für Zwecke der starken Normalisierung ist zunächst nicht klar, dass durch den Zwischenschritt (4) keine Divergenzen entstehen.) Wir verwenden die durch $A = \mathbb{N}_{\geqslant 2}$ und

$$p_\oplus(x, y) = x^2 + xy$$

gegebene Polynomordnung. Für Reduktion (4) rechnen wir

$$\begin{aligned} p_{(x\oplus y)\oplus z} &= p_\oplus(p_\oplus(x, y), z) \\ &= p_\oplus(x^2 + xy, z) \\ &= (x^2 + xy)^2 + (x^2 + xy)z \\ &= x^4 + 2x^3y + x^2y^2 + x^2z + xyz \end{aligned}$$

sowie

$$\begin{aligned} p_{x\oplus(y\oplus z)} &= p_\oplus(x, p_\oplus(y,z)) \\ &= x^2 + x(y^2 + yz) \\ &= x^2 + xy^2 + xyz, \end{aligned}$$

so dass in der Tat $p_{(x\oplus y)\oplus z} >_A p_{x\oplus(y\oplus z)}$ (warum?). Ferner haben wir

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

so dass offenbar $p_{x\oplus(y\oplus z)} >_A p_{y\oplus y}$. Achtung: In diesem Fall brauchen wir, dass die eingesetzten Werte $\geqslant 2$ sind, da sonst i.a. nicht $x^2 + xy^2 + xyz > 2y^2$.

Nach Korollar 2.44 ist $\to$ somit SN.

2.2.3 Konfluenz

Die im vorigen Abschnitt behandelte Terminierungsanalyse dient insbesondere der Sicherstellung der *Existenz* von Normalformen. Wir wenden uns als nächstes Kriterien für die *Eindeutigkeit* von Normalformen zu; der entscheidende Begriff hierbei ist der der *Konfluenz*. Wir illustrieren die möglichen Probleme hierbei zunächst an einem Beispiel.

Beispiel 2.46 (Gruppen)**.** Man erinnere sich, dass eine Gruppe definiert werden kann als eine Menge mit einer zweistelligen Operation $\cdot$, einem neutralen Element e und einer Inversenoperation $(-)^{-1}$, mit Gleichungen

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

Wir wandeln diese Gleichungen in ein TES um, indem wir oben einfach überall „=“ durch „$\to_0$“ ersetzen. Dann hat der Term $y \cdot (x \cdot x^{-1})$ zwei verschiedene Normalformen:

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

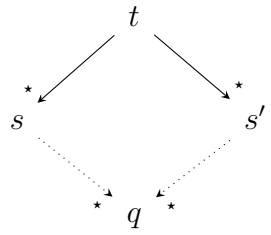
Wir werden nun Methoden kennenlernen, mit denen man solche Probleme systematisch lokalisieren bzw. ihre Abwesenheit zeigen kann.

Definition 2.47 (Diamant-Eigenschaft)**.** Eine Relation R hat die *Diamant-Eigenschaft*, wenn für all x, z, y mit xRy und xRz ein w mit zRw und zRw existiert.

Definition 2.48. Sei T ein Termersetzungssystem.

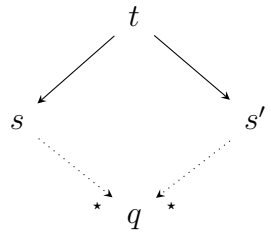
1. Terme s, s' heißen *zusammenführbar (zf.)*, wenn ein q existiert mit $s \to^\star q$ und $s' \to^\star q$.

2. Das Termersetzungssystem T heißt *konfluent* (*CR*, kurz für *Church-Rosser*), wenn für alle Terme t, s, s' mit $t \to^\star s$ und $t \to^\star s'$ die Terme s und s' zusammenführbar sind:



   – d.h. $\to^\star$ hat die Diamant-Eigenschaft.

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



Fakt 2.49. *Syntaktisch gleiche Terme sind zusammenführbar.*

Bemerkung 2.50. Konfluenz ist eine wichtige Eigenschaft insbesondere dann, wenn man Termersetzungssysteme als Programme ansieht, da sie Determinismus garantiert: Wenn ein Termersetzungssystem konfluent ist, mag es zwar eventuell mehrere mögliche Umformungswege geben, die jedoch alle (falls sie terminieren) zum gleichen Ergebnis führen. Wenn man (z.B. in der Computeralgebra und in (halb-)automatischen Beweisern) Termersetzung zur Analyse von Gleichungstheorien verwendet, folgt, wie wir sehen werden, aus CR und SN Entscheidbarkeit der durch das Termersetzungssystem dargestellten Gleichungstheorie.

Satz 2.51. *Sei T ein konfluentes Termersetzungssystem.*

1. *Für Terme t, s gilt*

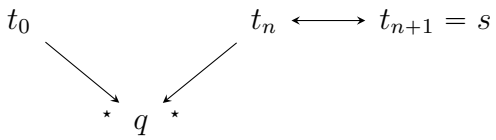
$$s \leftrightarrow^\star t \iff s \text{ und } t \text{ sind zusammenführbar.}$$

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

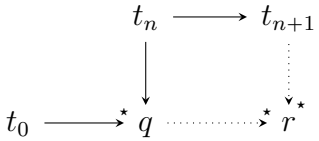
Beweis.

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

- $n = 0$: Dann gilt $s = t$, also sind t und s nach Fakt 2.49 zusammenführbar.
- $n \to n + 1$: Nach Induktionsvoraussetzung haben wir:



Fall 1: $t_{n+1} \to t_n$. Dann $t_{n+1} \to^\star q$, d.h. t_0 und $t_{n+1} = s$ sind zusammenführbar.
Fall 2: $t_n \to t_{n+1}$. Dann existiert per Konfluenz r mit $q \to^\star r$ und $t_{n+1} \to^\star r$. Es gilt dann auch $t = t_0 \to^\star r$, d.h. t und $t_{n+1} = s$ sind zusammenführbar:

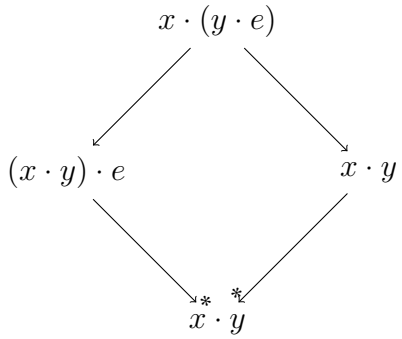


(2.) Nach Voraussetzung gilt $s \leftarrow^\star t \to^\star s'$, somit $s \leftrightarrow^\star s'$, also gibt es nach (1.) ein q mit $s \to^\star q \leftarrow^\star s'$, also $s = q = s'$, da s, s' normal sind. □

Beispiel 2.52 (Gruppen, Fortsetzung)**.** Wir erinnern an das Termersetzungssystem für Gruppen aus Beispiel 2.46:

$$x \cdot (y \cdot z) \to_0 (x \cdot y) \cdot z$$
$$x \cdot e \to_0 x$$
$$x \cdot x^{-1} \to_0 e.$$

Das obige Beispiel zeigt, dass hier lokale Konfluenz fehlschlägt; konkret reduziert $y \cdot (x \cdot x^{-1})$ sowohl auf y als auch auf $(y \cdot x) \cdot x^{-1}$, und diese Terme sind nicht zusammenführbar (da sie normal und verschieden sind). Wir finden auch positive Beispiele: Z.B. kann der Ausdruck $x \cdot (y \cdot e)$ zu zwei unterschiedlichen Ausdrücken umgeformt werden, die aber zusammenführbar sind:



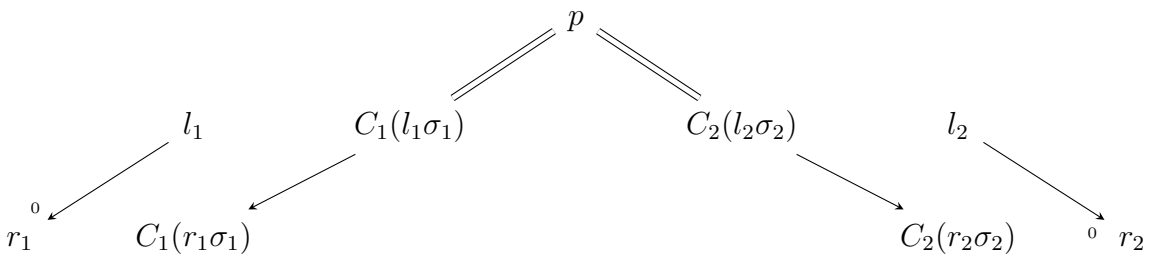
(natürlich ist $x \cdot y$ dann eine Normalform von $x \cdot (y \cdot e)$).

Die Bedeutung der *lokalen* Konfluenz liegt in der folgenden Tatsache:

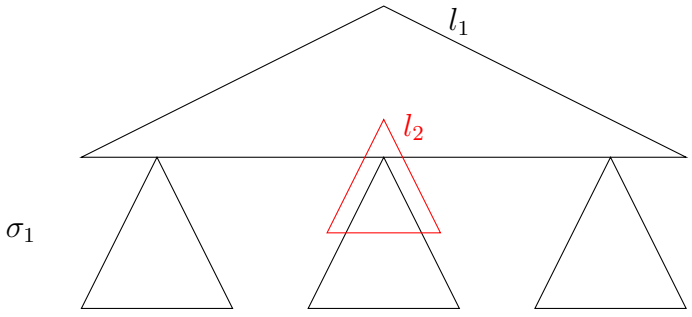
Satz 2.53 (Newman's Lemma)**.** *Ein stark normalisierendes und lokal konfluentes Termersetzungssystem ist konfluent. (Kurz: SN & WCR ⇒ CR.)*

Der Beweis folgt in Abschnitt 2.3.

Wir wenden uns nun der Frage zu, wann ein Termersetzungssystem lokal konfluent ist. Eine zentrale Rolle spielt hierbei der Begriff des *kritischen Paars* nach Knuth und Bendix. Betrachten wir die folgende Situation, in der ein Term p mit zwei verschiedenen Regeln reduziert werden kann:



Derartig konkurrierende Reduktionsmöglichkeiten werden insbesondere dann zum Problem, wenn (etwa) die Anwendung der zweiten Regel $l_2 \to_0 r_2$ im Termbaum unterhalb der von $l_1 \to_0 r_1$ liegt, also unter dem Kontext $C_1(\cdot)$, und somit durch ihre Anwendung die Anwendbarkeit der ersten Regel gestört wird. Die folgende Graphik 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 keine Substitutionsinstanz von l_1 mehr vor, $l_1 \to_0 r_1$ ist also nicht mehr anwendbar.



(6)

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

Definition 2.54. Eine Substitution σ heißt *Unifikator* von Termen t, s, wenn $t\sigma = s\sigma$. Wir setzen $Unif(t, s) := \{\sigma \mid \sigma \text{ ist Unifikator von } t, s\}$. Terme t, s heißen *unifizierbar*, wenn $Unif(t, s) \neq \emptyset$.

Beispiel 2.55.

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

Definition 2.56. Eine Substitution σ heißt *allgemeiner als* eine Substitution σ', wenn eine Substitution τ mit $\sigma' = \sigma\tau$ existiert (wobei $\sigma\tau$ weiterhin die Substitution ist, die auf den gleichen Variablen x wie σ definiert ist, und dort per $\sigma\tau(x) = \sigma(x)\tau$ agiert). Wir sagen, σ sei *allgemeinster Unifikator* von Termen t, s, und schreiben $\sigma = \mathsf{mgu}(t, s)$, wenn σ Unifikator von t, s ist sowie allgemeiner als alle Unifikatoren von t, s.

Wir erinnern uns aus GLoIn, dass der allgemeinste Unifikator eindeutig bis auf die Wahl der Variablennamen ist, und dass unifizierbare Terme stets einen allgemeinsten Unifikator besitzen. Diesen kann man über geeignete Algorithmen systematisch ausrechnen; in den Beispielen, die wir hier antreffen, sieht man ihn typischerweise mit bloßem Auge. Im positiven Beispiel oben ist z.B. der angegebene Unifikator klarerweise ein allgemeinster Unifikator.

In diesen Begriffen definieren wir kritischen Paare wie folgt.

Definition 2.57. Seien $l_1 \to_0 r_1$ und $l_2 \to_0 r_2$ zwei Umformungsregeln des Termersetzungssystems sowie $FV(l_1) \cap FV(l_2) = \emptyset$ (ggf. nach Umbenennung). Sei $l_1 = C(t)$, wobei t ein nichttrivialer Term ist (d.h. t ist nicht nur eine Variable), so dass t und l_2 unifizierbar sind. Sei $\sigma = \mathsf{mgu}(t, l_2)$. Dann heißt $(r_1\sigma, C(r_2)\sigma)$ ein *kritisches Paar*. Ein solches kritisches Paar ist *trivial*, wenn die beiden Umformungsregeln $l_1 \to_0 r_1$ und $l_2 \to_0 r_2$ bis auf Umbenennung gleich sind und $C(\cdot) = (\cdot)$.

Bemerkung 2.58. Wenn in den Bezeichnungen der obigen Definition $(r_1\sigma, r_2\sigma)$ ein triviales kritisches Paar ist, dann gilt offenbar bereits $r_1\sigma = r_2\sigma$, so dass triviale kritische Paare stets zusammenführbar sind; sie müssen in Aufgabenstellungen, die z.B. die Auflistung aller kritischen Paare eines Systems verlangen, nicht mit angegeben werden. Achtung: Es gibt auch nichttriviale kritische Paare, die aus zwei gleichen Termen bestehen. Wenn wir z.B. nur für Zwecke dieser Bemerkung das TES für Gruppen um die Regel $(x \cdot x^{-1}) \cdot x \to_0 e \cdot x$ erweitern, dann entsteht durch alternative Anwendung der neuen Umformungsregel und der alten Regel $x \cdot x^{-1} \to_0 e$ ein nichttriviales kritisches Paar $(e \cdot x, e \cdot x)$.

Lemma 2.59. *Wenn mit Bezeichnern wie in obiger Definition $(r_1\sigma, C(r_2)\sigma)$ ein kritisches Paar ist, dann gilt*

$$r_1\sigma \leftarrow l_1\sigma = C(l_2)\sigma \to C(r_2)\sigma \tag{7}$$

Beweis. Der erste Teil der Behauptung ist klar. Für die Gleichung rechnen wir wie folgt:

$$l_1\sigma = C(t)\sigma = C\sigma(t\sigma) = C\sigma(l_2\sigma) = C(l_2)\sigma,$$

wobei wir verwenden, dass für Terme s allgemein $C(s)\sigma = C\sigma(s\sigma)$ gilt, wie man leicht durch Induktion über $C(\cdot)$ zeigt. Ebenso sieht man die letzte Behauptung:

$$C(l_2)\sigma = C\sigma(l_2\sigma) \to C\sigma(r_2\sigma) = C(r_2)\sigma. \qquad \square$$

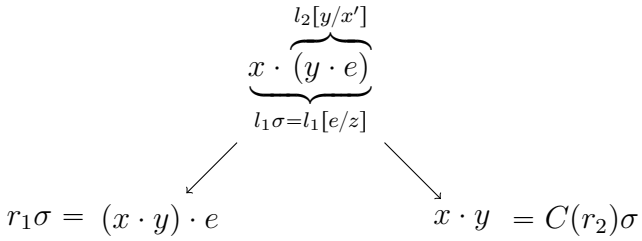
Beispiel 2.60 (Gruppen, siehe oben)**.** Wir vergeben Bezeichner für die in $\to_0$ vorkommenden Terme:

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

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

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

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



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

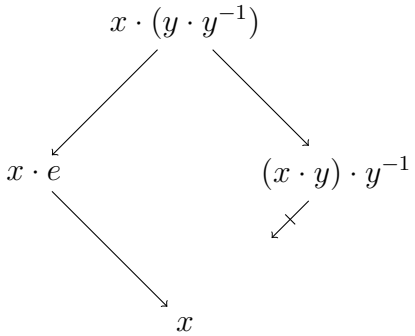
Beispiel 2.61. Wir wissen andererseits bereits, dass das TES für Gruppen in seiner jetzigen Form nicht lokal konfluent ist. Dies zeigt sich in der Tat an der Methode der kritischen Paare: Mit

$$l_1 \to_0 r_1 = (x \cdot (y \cdot z) \to_0 (x \cdot y) \cdot z)$$
$$l_2 \to_0 r_2 = (x' \cdot x'^{-1} \to_0 e)$$
$$C(\cdot) = x \cdot (\cdot)$$
$$t = y \cdot z$$

sind wieder $t = y \cdot z$ und $l_2 = x' \cdot x'^{-1}$ unifizierbar, mit mgu

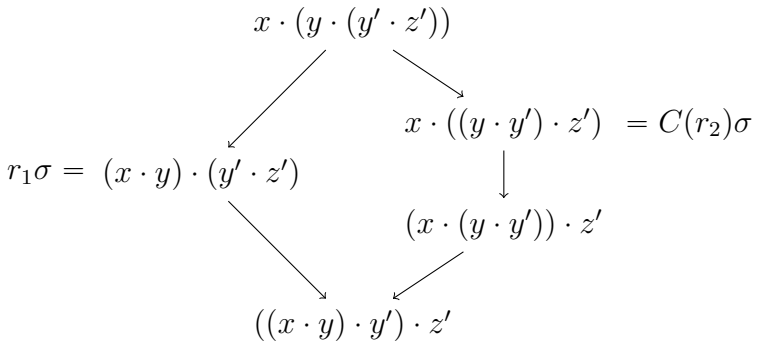
$$\sigma = \mathsf{mgu}(y \cdot z, x' \cdot x'^{-1}) = [y/x', y^{-1}/z],$$

und wir haben



d.h. wir sind auf ein nicht zusammenführbares kritisches Paar $(x \cdot e, (x \cdot y) \cdot y^{-1})$ gestoßen. Um ein konfluentes System zu erhalten, müssten wir also noch weitere Regeln hinzufügen, gemäß obiger Rechnung z.B. die Regel $(x \cdot y) \cdot y^{-1} \to_0 x$.

Beispiel 2.62. Achtung: die linke Seite einer Regel kann durchaus auch mit einem ihrer Teilterme unifizieren, was dann ein kritisches Paar ergibt. Z.B. können wir den Teilterm $y \cdot z$ der linken Seite $x \cdot (y \cdot z)$ des Assoziativgesetzes der linken Seite $x' \cdot (y' \cdot z')$ einer umbenannten Kopie unifizieren: Wir haben $\sigma = \mathsf{mgu}(y \cdot z, x' \cdot (y' \cdot z')) = [y/x', y' \cdot z'/z]$. Das entstehende kritische Paar ist zusammenführbar:



Bemerkung 2.63. Es gibt (für $\to_0$ endlich) nur endlich viele kritische Paare.

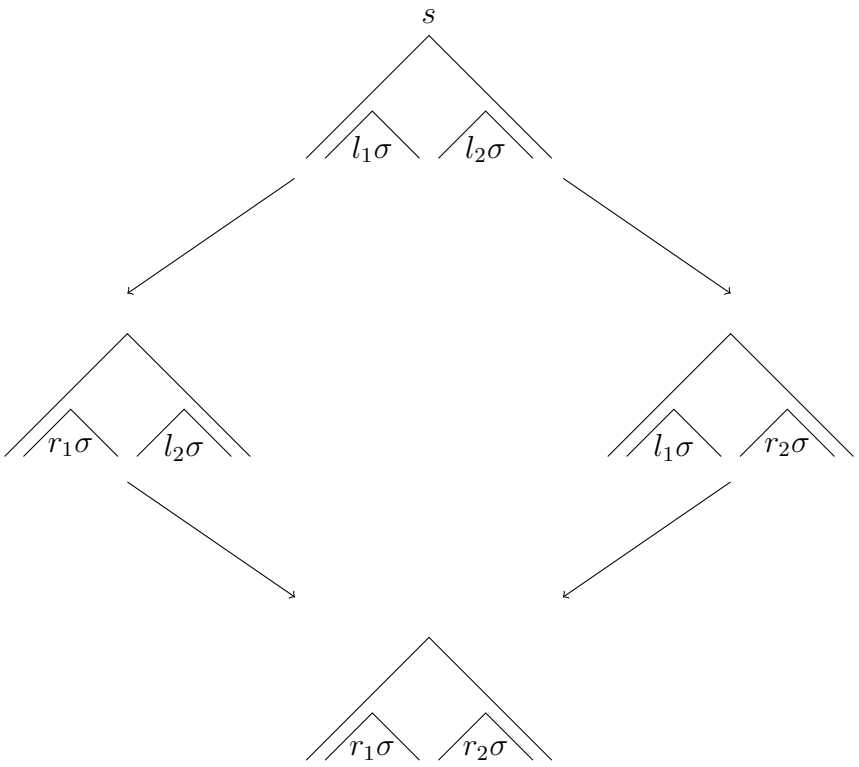
Satz 2.64 (Critical Pair Lemma)**.** *Ein Termersetzungssystem T ist genau dann lokal konfluent, wenn in T alle kritischen Paare zusammenführbar sind.*

Beweis. „⇒“ folgt sofort daraus, dass die Terme in einem kritischen Paar nach Lemma 2.59 Redukte eines gemeinsamen Ausgangsterms sind. Wir beweisen die umgekehrte Implikation „⇐“. Wir schreiben

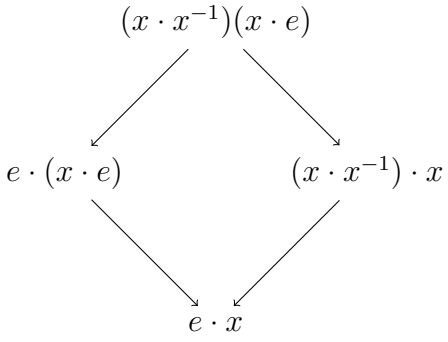
- $C(\cdot) \sqsubseteq D(\cdot)$, wenn $E(\cdot)$ existiert mit $C(\cdot) = D(E(\cdot))$, d.h. wenn $C(\cdot)$ aus $D(\cdot)$ durch Einsetzen hervorgeht.
- $C(\cdot) \perp D(\cdot)$, wenn $C(\cdot) \not\sqsubseteq D(\cdot)$ und $D(\cdot) \not\sqsubseteq C(\cdot)$.

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

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

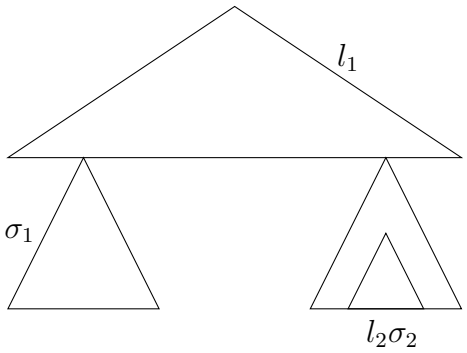


Beispiel: In unserem TES für Gruppen haben wir



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

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



O.E. nehmen wir $\sigma_2 = id$ an (ein nichttriviales σ_2 kann man sonst einfach hinterher wieder in die gesamte Rechnung einsetzen, was, wenn die Variablen in l_2 hinreichend frisch sind, den Rest des Terms nicht stört). Wir müssen zeigen, dass $C_2(r_2)$ und $r_1\sigma_1$ zusammenführbar sind. Wir haben gesehen, dass r_2 innerhalb eines Teilterms $\sigma_1(x)$ liegt; wenn x in l_1 mehrfach vorkommt, ist die Regel $l_1 \to_0 r_1$ jetzt zunächst im Allgemeinen nicht mehr anwendbar. Wir stellen Anwendbarkeit von $l_1 \to_0 r_1$ wieder her, indem wir $l_2 \to_0 r_2$ auch für alle anderen Vorkommen von x auf den entsprechenden Teilterm $\sigma_1(x)$ in $C_2(r_2)$ anwenden. Wir erreichen dann wieder einen Term der Form $l_1\sigma''$, auf den wir die erste Regel anwenden können. Auf $r_1\sigma_1$ können wir sofort die zweite Regel anwenden (wiederum so oft, wie x in r_1 vorkommt), da ja l_2 innerhalb von σ_1 liegt; damit erreichen wir denselben Term wie vorher, haben also Konfluenz gezeigt. Abbildung 2 illustriert den Fall, dass x in l_1 dreimal und in r_1 zweimal vorkommt.

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

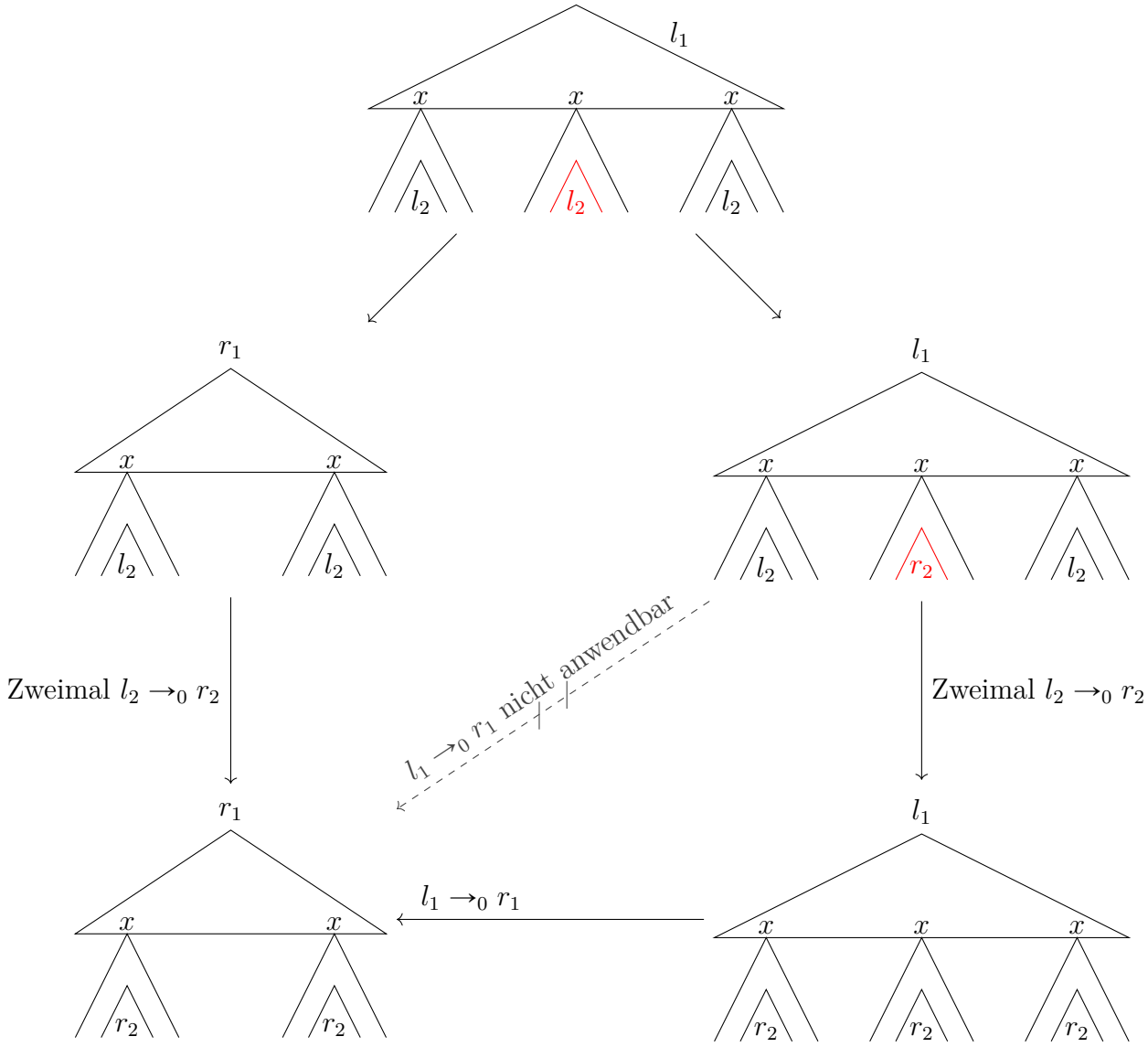


Abbildung 2: Konfluenz in Fall 2a

2.3 Wohlfundierte Induktion

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

Satz 2.65. *Sei $R \subseteq X \times X$ eine wohlfundierte Relation auf einer Menge X. Dann gilt folgendes Prinzip der* wohlfundierten Induktion*:*
Falls gilt: $\forall x.\, (\forall y.\, x\, R\, y \Rightarrow P(y)) \Rightarrow P(x) \qquad (*)$
Dann folgt: $\forall x.\, P(x)$

Dabei nennen wir die Implikation $(*)$ den *Induktionsschritt*, die im Induktionsschritt verwendete Annahme $P(y)$ für alle y mit xRy die *Induktionsvoraussetzung* (kurz: *IV*) und das im Induktionsschritt bewiesene Sukzedens $P(x)$ die *Induktionsbehauptung*.

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

Bemerkung 2.66. Der obige Beweis verwendet ein (allerdings relativ harmloses) mengentheoretisches Auswahlprinzip, *dependent choice*.

Beispiel 2.67. Aus wohlfundierter Induktion folgen alle anderen uns bekannten Induktionsprinzipien:

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

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

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

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

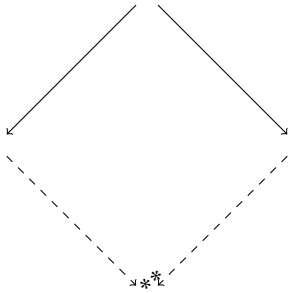
3. Wir definieren eine Relation $R \subseteq T_\Sigma(V) \times T_\Sigma(V)$ auf Termen durch

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

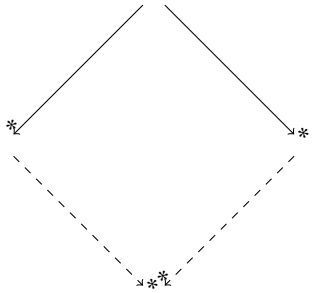
   d.h. tRs genau dann, wenn s ein unmittelbarer Unterterm von t ist. Die Relation R ist wohlfundiert; wohlfundierte Induktion über R ist *strukturelle Induktion* über Terme: Wenn für jede Operation $f/n \in \Sigma$ aus $P(t_1) \wedge \cdots \wedge P(t_n)$ stets $P(f(t_1, \ldots, t_n))$ folgt, dann gilt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

Beweis von Newmans Lemma (Satz 2.53). (D.h. T stark normalisierend und lokal konfluent $\Rightarrow T$ konfluent.)

Erinnerung:

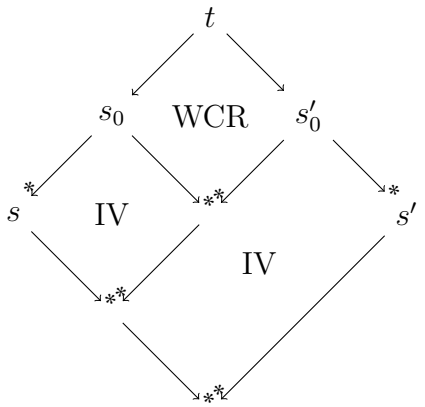


lokale Konfluenz



Konfluenz

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



□

3 Der λ-Kalkül

- Funktionales Programmieren mit höheren Funktionen:

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

- Ungetypter Lambdakalkül = Lisp
- Slogan: λ-Kalkül = Kalkül der anonymen Funktionen
- Geht zurück auf Church, Curry, Kleene, Rosser.

3.1 Der ungetypte λ-Kalkül

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

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

$$\lambda x.\, t$$

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

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

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

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

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

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

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

Notation 3.2. Wir verwenden folgende Konventionen:

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

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

- $FV(x) = \{x\}$ (für $x \in V$)
- $FV(ts) = FV(t) \cup FV(s)$
- $FV(\lambda x.\,t) = FV(t) \backslash \{x\}$

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

Definition 3.4 (Substitution)**.** Eine Substitution ist (im wesentlichen wie bisher) eine Abbildung $\sigma : V_0 \to T(V)$, die Variablen aus einer endlichen Teilmenge $V_0 \subseteq V$ auf Terme abbildet. Die Anwendung einer Substitution σ auf Terme des λ-Kalküls ist rekursiv definiert durch

- $x\sigma = \sigma(x)$, wenn $x \in V_0$, und sonst $x\sigma = x$
- $(ts)\sigma = (t\sigma)(s\sigma)$
- $(\lambda x.t)\sigma = \lambda y.(t\sigma')$, wobei y eine frische Variable ist (also $y \notin FV(\sigma(z))$ für alle $z \in FV(\lambda x.\,t)$) und $\sigma' = \sigma[x \mapsto y]$ (d.h. σ' bildet x auf y ab und verhält sich ansonsten wie σ).

(Wenn in der letzten Klausel x die genannte Frischheitsbedingung an y erfüllt, kann $y = x$ gewählt werden.)

Bemerkung 3.5. Die Bedingung $y \notin FV(\sigma(z))$ in der letzten Klausel dient, wie bereits in GLoIn bei der Substitution in Formeln mit Quantoren, zur Vermeidung eines *Variableneinfangs* (*variable capture*), d.h. der Substitution einer vorher freien Variablen an eine Stelle, an der sie dann gebunden würde. Z.B. sollte vernünftigerweise *nicht* $\lambda x.\,x$ durch Substitution von x für y in $\lambda x.\,y$ entstehen – d.h. die Identitätsfunktion sollte keine Substitutionsinstanz von „konstante Funktion mit Wert y“ sein. Trotzdem wollen wir natürlich jede Substitution auf jeden Term anwenden; deswegen weichen wir dem Variableneinfang durch Umbenennen gebundener Variablen aus (*capture-avoiding substitution*). Substitution ist so dann allerdings nur bis auf Umbenennung gebundener Variablen definiert, d.h. bis auf *α-Äquivalenz* im als nächstes definierten Sinn.

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

$$\lambda x.t =_\alpha \lambda y.t[y/x] \qquad \text{wenn } y \notin FV(\lambda x.t). \tag{8}$$

erzeugte Kongruenz.

(In Umbenennungen gemäß (8) darf die links gebundene Variable x also in keine im Funktionskörper t frei vorkommende Variable umbenannt werden außer in x selbst; in letzterem Fall sind natürlich dann die beiden Seiten schon gleich. Wir lassen diesen Fall aus technischen Gründen zu, um den Beweis von Lemma 3.8 zu vereinfachen.)

Beispiel 3.7. Es gilt $\lambda x.\, xy =_\alpha \lambda z.\, zy$, aber $\lambda x.\, xy \neq_\alpha \lambda y.\, yy$.

Lemma 3.8. *$=_\alpha$ ist stabil.*

Beweis. Man prüft allgemein leicht nach, dass die von einer stabilen Relation erzeugte Kongruenz wieder stabil ist. Es reicht also zu zeigen, dass die Relation

$$R = \{(\lambda x.t, \lambda y.t[y/x]) \mid t \text{ Term}, y \notin FV(\lambda x.t)\}$$

stabil ist. Seien also t ein Term, $y \notin FV(\lambda x.t)$ (und somit $(\lambda x.t)\, R\, (\lambda y.t[y/x])$) und σ eine Substitution. Wir müssen zeigen, dass

$$(\lambda x.t)\sigma\, R\, (\lambda y.t[y/x])\sigma \tag{*}$$

gilt. Nach der obigen Definition der Anwendung von σ ist die linke Seite $\lambda x'.t\sigma'$ für frisches x' und $\sigma' = \sigma[x \mapsto x']$, und die rechte Seite $\lambda y'.t[y/x]\sigma''$ für ein weiteres frisches y' und $\sigma'' = \sigma[y \mapsto y']$. Explizit bedeutet 'frisch' hier, dass $x' \notin FV(\sigma(z))$ für alle $z \in FV(\lambda x.t)$, bzw. dass $y' \notin FV(\sigma(z))$ für alle $z \in FV(\lambda y.t[y/x])) = FV(\lambda x.t)$. Wir rechnen nach, dass die Substitutionen $[y/x]\sigma''$ und $\sigma'[y'/x']$ auf freien Variablen $z \in FV(t)$ übereinstimmen:

- $z = x$: $([y/x]\sigma'')(z) = \sigma''(y) = y'$, $(\sigma'[y'/x'])(z) = x'[y'/x'] = y'$.
- $z = y$: Da $z \in FV(t)$ und $y \notin FV(\lambda x.t)$, kann dies nur vorkommen, wenn $y = x$, so dass dieser Fall schon erledigt ist.
- $z \notin \{x, y\}$: Wir haben $([y/x]\sigma'')(z) = \sigma''(z) = \sigma(z)$ und $\sigma'[y'/x'](z) = \sigma(z)[y'/x'] = \sigma(z)$, da x' frisch war und damit insbesondere $x' \notin FV(\sigma(z))$, da $z \in FV(\lambda x.t)$ (wegen $z \in FV(t)$ und, im aktuellen Fall, $z \neq x$).

Die rechte Seite in (*) ist also gleich $\lambda y'.(t\sigma')[y'/x']$, und somit wie verlangt unter R in Relation zur linken Seite $\lambda x'.\, t\sigma'$, da ferner $y' \notin FV(\lambda x'.t\sigma')$. Um Letzteres zu sehen, argumentieren wir wie folgt: Wäre $y' \in FV(\lambda x'.t\sigma')$, dann hätten wir $z \in FV(\lambda x.t)$ mit $y' \in FV(\sigma'(z))$. Per Definition von σ' gilt $\sigma'(z) = \sigma[x \mapsto x'](z) = \sigma(z)$, da $z \neq x$. Damit $y' \in FV(\sigma(z))$, was der Frische von y' widerspricht, da $z \in FV(\lambda y.\, t[y/x]) = FV(\lambda x.t)$. □

3.1.1 β-Reduktion

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

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

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

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

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

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

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

Beispiel 3.9.

- $(\lambda x.\, xx)(yx) \to_\beta (yx)(yx) = yx(yx)$
- Capture avoidance: $(\lambda x.\, \lambda y.\, x)\, y \to_\beta \lambda y'.\, y$
- Nichtterminierung: wir schreiben $\omega = \lambda x.\, xx$. Dann haben wir

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

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

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

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

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

3.1.2 Rekursion

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

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

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

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

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

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

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

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

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

Satz 3.10. *Im ungetypten λ-Kalkül*

1. *hat jeder Term t einen* Fixpunkt*, d.h. einen Term s mit $s \to_\beta ts$.*
2. *gibt es einen* Fixpunktkombinator *Y, d.h. für jeden Term t haben wir $Y\,t \to_\beta s$ für einen Fixpunkt s von t im obigen Sinn.*

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

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

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

Korollar 3.11. *Für jeden Term $t[f, x]$ mit freien Variablen f, x existiert ein Term s mit $s\,x \to_\beta^\star t[s, x]$.*

(Dabei ist $t[s, x]$ einfach eine lesbarere Schreibweise für $t[s/f]$.) Mit anderen Worten können wir rekursive Definitionen der Form $f\,x = t[f, x]$ schreiben, die rechts sowohl die rekursiv definierte Funktion f als auch ihr Argument x erwähnen.

Beweis. Nach dem vorigen Satz hat der Term $\lambda f.\, \lambda x.\, t[f, x]$ einen Fixpunkt, d.h. wir haben einen Term s mit $s \to_\beta (\lambda f.\, \lambda x.\, t[f, x])\,s$. Dann gilt

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

3.1.3 Auswertungsstrategien und Standardisierung

Die Reihenfolge der Reduktion von β-Redexen kann erheblichen Einfluss auf das Terminierungsverhalten haben:

Beispiel 3.12. Ob der Ausdruck

$$(\lambda xy.\, y)(\omega\omega)y$$

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

Konkrete Programmiersprachen legen, wie oben angedeutet, oft eine konkrete Auswertungsstrategie fest. Wichtig sind unter anderem die folgenden Varianten:

Definition 3.13. Die *applikative (auch: leftmost-innermost) Reduktion* $\to_a$ ist induktiv definiert durch:

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

Als Herleitungsregeln:

$$\frac{}{(\lambda x.\, t)s \to_a t[s/x]}\ (s \text{ und } t \text{ normal}) \qquad \frac{t \to_a t'}{\lambda x.\, t \to_a \lambda x.\, t'}$$

$$\frac{t \to_a t'}{ts \to_a t's} \qquad \frac{s \to_a s'}{ts \to_a ts'}\ (t \text{ normal})$$

Definition 3.14. Die *normale (auch: leftmost-outermost) Reduktion* $\to_n$ ist definiert durch

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

Als Herleitungsregeln:

$$\frac{}{(\lambda x.\, t)s \to_n t[s/x]} \qquad \frac{t \to_n t'}{\lambda x.\, t \to_n \lambda x.\, t'} \qquad \frac{t \to_n t'}{ts \to_n t's}\ (t \text{ keine } \lambda\text{-Abstraktion})$$

$$\frac{s \to_n s'}{ts \to_n ts'}\ (t \text{ normal und keine } \lambda\text{-Abstraktion})$$

Hierbei bedeutet *outermost*, dass immer ein ganz außen liegender β-Redex reduziert wird, der also nicht innerhalb eines anderen β-Redex liegt. Dies spiegelt sich in der Einschränkung wieder (in Definition 3.14), dass in ts nur dann s oder t reduziert werden, wenn t keine λ-Abstraktion ist. Wenn es mehrere ganz außen liegende β-Redexe gibt, wird der am weitesten links liegende reduziert; dies äußert sich in der Einschränkung, dass in ts nur dann s reduziert wird, wenn t schon normal ist (sowohl in Definition 3.14 als auch in Definition 3.13). Umgekehrt heißt *innermost*, dass nur solche β-Redexe reduziert werden, die ganz innen liegen, also keine weiteren β-Redexe enthalten, was durch die Einschränkung (in Definition 3.13) bewirkt wird, dass $(\lambda x.\, t)\, s$ nur dann reduziert wird, wenn sowohl t als auch s normal sind, also keine β-Redexe enthalten.

Man kann die obigen Definitionen auch mittels Reduktionsalgorithmen verstehen, die den Termbaum von der Wurzel aus traversieren: *leftmost* bedeutet, dass immer der linke Teilbaum (unter einer Applikation) zuerst durchsucht wird; bei *outermost* wird jeder gefundene Redex sofort reduziert, währen bei *innermost* nur dann reduziert wird, wenn keine weiter unten liegenden Redexe mehr gefunden werden. (In beiden Fällen beginnt dann übrigens die Suche im nächsten Reduktionsschritt wieder an der Wurzel.)

Die Bedeutung von *leftmost* ist dabei relativ untergeordnet; es bewirkt im wesentlichen nur, dass bei Anwendung von Funktionen auf mehrere Argumente, also in Termen der Form $t\, s_1 \ldots s_n$, die Argumente $s_1, \ldots, s_n$ von links nach rechts ausgewertet werden (sobald die Reduktion von Funktionsargumenten denn gemäß der Strategie an der Reihe ist). Konkreter nehmen wir an, dass t keine Applikation ist (sonst gibt es einfach ein s_i mehr). Dann kann t entweder eine Variable sein, was in beiden Strategien bewirkt, dass als nächstes die Argumente reduziert werden, beginnend bei s_1, oder eine λ-Abstraktion $\lambda x.\, t'$. Bei normaler Reduktion wird dann zuerst der β-Redex $(\lambda x.\, t')\, s_1$ reduziert, während bei applikativer Reduktion zuerst t' ausreduziert wird, mit Ergebnis t'', und anschließend s_1, mit Ergebnis s_1', woraufhin der dann entstehende β-Redex $(\lambda x.\, t'')\, s_1'$ reduziert wird. Nur an der letzten Stelle unterscheidet sich *rightmost* (das wir hier nicht weiter behandeln) substanziell von *leftmost*: Unter *rightmost* würden in jedem Fall zuerst die Argumente s_i mit $i \geqslant 2$ ausreduziert, beginnend bei s_n, und dann erst gemäß der jeweiligen Strategie der Term $(\lambda x.\, t'')\, s_1$.

Lemma 3.15. *Die applikative Reduktion ist deterministisch, d.h. wenn $t \to_a u$ und $t \to_a u'$, dann $u = u'$; entsprechendes gilt für normale Reduktion $\to_n$.*

Beweis. Applikative Reduktion: Induktion über t. Wir unterscheiden folgende Fälle:

- Variable x: Kommt unter den gegebenen Voraussetzungen nicht vor, da Variablen nicht reduzieren.
- $\lambda x.t$: Reduktionen sind nur über die Regel für λ-Abstraktionen herleitbar, und dann der Form $\lambda x.\, t \to_a \lambda x.\, t'$ mit $t \to_a t'$. Nach IV ist dabei t' durch t eindeutig bestimmt.
- ts: Wir unterscheiden die folgenden Fälle:
  - t und s sind normal. Dann ist $t = \lambda x.\, t_0$, und die Reduktion ist eindeutig bestimmt als $(\lambda x.\, t_0)s \to_a t_0[s/x]$.

- t is normal, aber nicht s. Dann ist die Reduktion von der Form $ts \to_a ts'$ mit $s \to_a s'$. Nach IV ist s' dabei eindeutig bestimmt, also auch der Term ts'.
- t ist nicht normal. Dann ist die Reduktion von der Form $ts \to_a t's$ mit $t \to_a t'$. Nach IV ist t' dabei eindeutig bestimmt, also auch der Term $t's$.

Der Beweis für normale Reduktion ist ähnlich. □

Bemerkung 3.16. Die applikative Reduktion führt im Beispiel 3.12 $((\lambda xy.\, y)(\omega\omega)y)$ zu Nichtterminierung, während die normale (leftmost-outermost) Reduktion terminiert. Wir werden sehen, dass normale Reduktion immer terminiert, *wenn* es eine Normalform gibt.

Dennoch ist es oft sinnvoll, applikative Reduktion zu bevorzugen – grob gesagt beruhen Beispiele, in denen normale Reduktion terminiert, applikative aber nicht, darauf, dass eine Funktion ein nichtterminierendes Argument gar nicht auswertet. Das kommt zwar vor; häufiger ist aber der Fall, dass ein Argument *mehrmals* verwendet wird, und dann unter normaler Reduktion auch mehrmals ausgewertet wird. Z.B. verdoppelt sich im Term

$$(\lambda x.\, fxx)t$$

der (womöglich sehr komplizierte) Term t bei normaler Reduktion, und wird dann auch zweimal ausgewertet, während t bei applikativer Reduktion nur einmal ausgewertet wird.

Definition 3.17. Eine Reduktion $t \to_\beta^\star s$ heißt *erfolgreich*, wenn sie in einer Normalform s endet.

Satz 3.18 (Normalisierungssatz)**.** *Jede erfolgreiche Reduktion kann durch eine normale Reduktion ersetzt werden. In Formeln: $t \to_\beta^\star s$, s Normalform $\Rightarrow t \to_n^\star s$.*

Wir präsentieren als nächstes den erstaunlich einfachen Beweis des Normalisierungssatzes nach Kashima. Wir benötigen Begriffe von *Weak Head Reduction* und *Standardreduktion*.

Definition 3.19 (Weak Head Reduction)**.** *Weak Head Reduction* $\to_w$: Wir können jeden Term t in der Form $t = t_1 \dots t_n$ schreiben, so dass t_1 entweder eine Variable oder eine λ-Abstraktion (aber keine Applikation) ist. Dann arbeitet Weak Head Reduction wie folgt:

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

Offensichtlich gilt $\to_w \subseteq \to_n$.

Definition 3.20 (Standardreduktion)**.** *Standardreduktion* $\to_{sta}$ ist definiert durch

1. $s \to_{sta} x$, wenn $s \to_w^\star x$, mit x Variable.
2. $s \to_{sta} tr$, wenn $s \to_w^\star t'r'$, $t' \to_{sta} t$ und $r' \to_{sta} r$.
3. $s \to_{sta} \lambda x.\, t$, wenn $s \to_w^\star \lambda x.\, t'$ und $t' \to_{sta} t$.

Anders als für normale Reduktion ist die Standardreduktion eines gegebenen Terms s im allgemeinen nicht eindeutig. Nehmen wir als Beispiel $s = (\lambda x.\, x)((\lambda y.\, y)x)$. Dann haben wir $s \to_w (\lambda y.\, y)x \to_w x$ und daher $s \to_{sta} x$, aber auch $s \to_{sta} (\lambda x.\, x)x$, weil $s \to_w^\star s$. Wenn aber t normal ist, dann kann man relativ direkt die Reduktion $s \to_{sta} t$ in eine Reduktion $s \to_n^\star t$ umwandeln. Wir beweisen das wie folgt.

Lemma 3.21. *Sei $s \to_{sta} t$ und t normal. Dann $s \to_n^\star t$.*

Beweis. Induktion über die Herleitung von $\to_{sta}$.

1. $t = x$ und $s \to_w^\star x$: offensichtlich.

2. $t = pq$, $s \to_w^\star p'q'$, $p' \to_{sta} p$ und $q \to_{sta} q$: Da pq normal ist, sind p und q ebenfalls normal, und p ist keine λ-Abstraktion. Per Induktionsvoraussetzung folgen $p' \to_n^\star p$ und $q' \to_n^\star q$. Da p keine λ-Abstraktion ist, sind auch alle Terme, von denen p per β-Reduktion erreichbar ist, keine λ-Abstraktionen, insbesondere also alle Terme in der Reduktionssequenz $p' \to_n^* p$. Somit bekommen wir

$$s \to_w^\star p'q' \to_n^\star pq' \to_n^\star pq.$$

3. $t = \lambda x.\, r$, $s \to_w^\star \lambda x.\, r'$ und $r' \to_{sta} r$: Da $\lambda x.\, r$ normal ist, ist r ebenfalls normal. Per Induktionsvoraussetzung folgt $r' \to_n^\star r$, und wir sind fertig per Definition von $\to_n^\star$. □

Wir untersuchen anschließend das Verhältnis zwischen $\to_w$ und $\to_{sta}$. Als Vorbereitungsschritt brauchen wir die folgende Eigenschaft aufeinanderfolgender Substitutionen:

Lemma 3.22 (Syntaktisches Substitutionslemma)**.** *Wenn $y \neq x$ und $y \notin FV(s)$, dann*

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

Dieses Lemma besagt insbesondere, dass die Einschrittreduktion $\to$ des λ-Kalküls tatsächlich stabil ist (für den Spezialfall von Substitutionen, die nur eine Variable ersetzen; der allgemeine Fall ist ähnlich): O.E. nehmen wir die Voraussetzungen des Lemmas an und haben dann

$$((\lambda y.\, t)\, u)[s/x] = (\lambda y.\, t[s/x])\, u[s/x] \rightrightarrows t[s/x][u[s/x]/y] = t[u/y][s/x].$$

Beweis (syntaktisches Substitutionslemma). Induktion über Struktur von t:
$t = y$: dann $y[u/y][s/x] = u[s/x]$ und $y[s/x][u[s/x]/y] = y[u[s/x]/y] = u[s/x]$. (Letzteres verwendet in der ersten Umformung die Annahme $y \neq x$.)
$t = x$: dann $x[u/y][s/x] = s$ (da $y \neq x$) und $x[s/x][u[s/x]/y] = s[u[s/x]/y] = s$ (da $y \notin FV(s)$).
Die restlichen Fälle ($t = z \notin \{y, x\}$, $t = t_1 t_2$, $t = \lambda w.\, t_0$) sind leicht. □

Lemma 3.23.

1. $s \to_{sta} s$.

2. *Wenn* $s \to_w t$, *dann* $sr \to_w tr$.
3. *Wenn* $s \to_w t \to_{sta} r$, *dann* $s \to_{sta} r$.
4. *Wenn* $s \to_w t$, *dann* $s[r/x] \to_w t[r/x]$.
5. *Wenn* $s \to_{sta} t$ *und* $p \to_{sta} q$, *dann* $s[p/x] \to_{sta} t[q/x]$.

Beweis.

1. Direkt per Induktion über s.
2. Klar nach Definition.
3. Klar nach Definition.
4. Folgt aus dem syntaktischen Substitutionslemma (Lemma 3.22).
5. Induktion über die Herleitung von $s \to_{sta} t$ anhand von (4) und (3). □

Lemma 3.24. *Wenn* $s \to_{sta} C((\lambda x.\, t)r) \to_\beta C(t[r/x])$, *dann* $s \to_{sta} C(t[r/x])$.

Beweis. Wir behandeln zunächst den Fall, dass C der leere Kontext ist. Nach Definition von $\to_{sta}$ haben wir dann p und q mit $s \to_w^\star pq$, $p \to_{sta} \lambda x.\, t$ und $q \to_{sta} r$. Wiederum per Definition von $\to_{sta}$ (für $p \to_{sta} \lambda x.\, t$) haben wir ferner dann t' mit $p \to_w^\star \lambda x.\, t'$ und $t' \to_{sta} t$. Daraus folgt

$$\begin{aligned}
s &\to_w^\star pq \\
&\to_w^\star (\lambda x.\, t')q && \text{(mit } p \to_w^\star \lambda x.\, t' \text{ und Lemma 3.23 (2))} \\
&\to_w t'[q/x] && \text{(Definition von } \to_w\text{)} \\
&\to_{sta} t[r/x], && \text{(mit } t' \to_{sta} t,\ q \to_{sta} r \text{ und Lemma 3.23 (5))}
\end{aligned}$$

und damit $s \to_{sta} t[r/x]$ nach Lemma 3.23 (3).

Den allgemeinen Fall handeln wir per Induktion über C ab:

- $C = (\cdot)$: Siehe oben.
- $C = Dq$: Nach Definition von $\to_{sta}$ haben wir p und q' mit $s \to_w^\star pq'$, $p \to_{sta} D((\lambda x.\, t)r)$ und $q' \to_{sta} q$. Nach Induktionsvoraussetzung folgt $p \to_{sta} D(t[r/x])$, und daher $s \to_{sta} D(t[r/x])q = C(t[r/x])$.
- $C = pD$: analog.
- $C = \lambda y.\, D$: Nach Definition von $\to_{sta}$ haben wir s' mit $s \to_w^\star \lambda y.\, s'$ und $s' \to_{sta} D((\lambda x.\, t)r)$. Nach Induktionsvoraussetzung folgt $s' \to_{sta} D(t[r/x])$, und daher $s \to_{sta} \lambda y.\, D(t[r/x]) = C(t[r/x])$. □

Korollar 3.25. *Wenn* $s \to_\beta^\star t$, *dann* $s \to_{sta} t$.

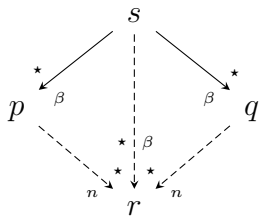
Beweis. Nach Lemma 3.23 (1) haben wir $s \to_{sta} s \to_\beta^\star t$, und mit Lemma 3.24 erhalten wir induktiv $s \to_{sta} t$. □

Der Beweis des Normalisierungssatzes läuft dann wie folgt. Sei $s \to_\beta^\star t$ eine erfolgreiche Reduktion. Nach Korollar 3.25 folgt $s \to_{sta} t$, und daher $s \to_n^\star t$ nach Lemma 3.21.

3.1.4 Church-Rosser im λ-Kalkül

Wir zeigen nunmehr, dass β-Reduktion im ungetypten λ-Kalkül konfluent ist. Dabei steht uns Newman's Lemma nicht zur Verfügung, da der ungetypte λ-Kalkül ja nicht stark normalisierend ist (noch nicht einmal schwach).

In Anbetracht des Normalisierungssatzes sind Terme, die keine Normalform haben, die einzigen, die einer besonderen Behandlung bedürfen. Wenn nämlich s eine Normalform r hat und $s \to_\beta^\star p$, dann gilt $p \to_n^\star r$ (das kann man aus dem Normalisierungssatz folgern, was wir hier aber nicht durchführen). Da r nur von s abhängt, nicht aber von p, erhalten wir damit in diesem Fall sofort Zusammenführbarkeit:



Dieses Ergebnis beruht abstrakter gesehen darauf, dass der gegebene Term r mit $s \to_\beta^* r$ von s aus mit einer deterministischen Auswertungsstrategie, wie eben der normalen Reduktion, erreichbar ist (so dass r im Endeffekt nur von s abhängt, und nicht etwa von p). In Verbindung mit der normalen Reduktion ist dieser Ansatz jedoch nicht für solche s geeignet, die keine Normalform haben. Zum Beispiel haben wir

$$s = (\omega\omega)((\lambda x.\, x)x) \to_\beta (\omega\omega)(x) = s',$$

aber sowohl s als auch s' reduzieren unter normaler Reduktion jeweils nur zu sich selbst. Das Problem liegt natürlich darin, dass die normale Reduktion hier nur den jeweils am weitesten links liegenden Redex reduziert, ohne jemals andere Redexe zu erreichen. Um die Konfluenzeigenschaft zu beweisen, verwenden wir daher eine andere Strategie, die in der Lage ist, viele Redexe gleichzeitig zu reduzieren, egal wie tief sie im gegebenen Term vorkommen:

Wir definieren die *parallele (und geschachtelte) Reduktion* $\rightrightarrows$ induktiv durch

a) $x \rightrightarrows x$

b) $\lambda x.\, t \rightrightarrows \lambda x.\, s$, wenn $t \rightrightarrows s$

c) $ts \rightrightarrows t's'$, wenn $t \rightrightarrows t'$ und $s \rightrightarrows s'$

d) $(\lambda x.\, t)s \rightrightarrows t'[s'/x]$, wenn $t \rightrightarrows t'$ und $s \rightrightarrows s'$

– d.h. die parallele Reduktion reduziert gleichzeitig beliebig viele β-Redexe in einem Term (eventuell mehrfach, wenn eine Reduktion mehrere Kopien eines Unterterms einführt). Man beachte, dass aus den ersten drei Klauseln per Induktion über t folgt, dass

$$t \rightrightarrows t \text{ für jeden Term } t, \tag{9}$$

d.h. die parallele Reduktion enthält auch die Option, Reduktionen zu unterlassen.

Die induktive Definition bedeutet wie stets, dass $t \rightrightarrows s$ genau dann gilt, wenn sich dies mit den oben angegebenen Regeln in endlich vielen Schritten herleiten lässt.

Lemma 3.26. *Die transitive und reflexive Hülle der parallele Reduktion ist gleich der der β-Reduktion: $\rightrightarrows^\star = \to_\beta^\star$.*

Beweis. „$\subseteq$“*:* Man zeigt leicht per Induktion über die Herleitung, dass aus $t \rightrightarrows s$ folgt, dass $t \rightrightarrows^\star s$; es folgt sofort $\rightrightarrows^\star \subseteq \to_\beta^\star$.

„$\supseteq$“*:* Wegen (9) kann man in einer parallele Reduktion auch nur einen einzigen β-Redex reduzieren, d.h. man hat $\to_\beta \subseteq \rightrightarrows$ und dann natürlich auch $\to_\beta^\star \subseteq \rightrightarrows^\star$. □

Wir halten folgende Eigenschaft fest, die noch einmal unsere Intuition bestätigt, dass parallele Reduktion parallel funktioniert:

Lemma 3.27. *Wenn $t \rightrightarrows t'$ und $s \rightrightarrows s'$, dann $t[s/x] \rightrightarrows t'[s'/x]$.*

Der Beweis ist durch einfache Induktion über die Herleitung von $t \rightrightarrows t'$, braucht allerdings im Induktionssschritt für die Regel (d) Lemma 3.22.

Beweis (Lemma 3.27). Wie angedeutet führen wir nur den Induktionsschritt für Regel (d) durch: Sei also $(\lambda y.\, t)\, u \rightrightarrows t'[u'/y]$ mit $t \rightrightarrows t'$ und $u \rightrightarrows u'$. Per Induktionsvoraussetzung haben wir $t[s/x] \rightrightarrows t'[s'/x]$ und $u[s/x] \rightrightarrows u'[s'/x]$. Wir nehmen o.E. an, dass $y \notin FV(s)$ und $y \neq x$. Dann haben wir wie verlangt

$$((\lambda y.\, t)u)[s/x] = (\lambda y.\, t[s/x])(u[s/x]) \rightrightarrows t'[s'/x][u'[s'/x]/y] = t'[u'/y][s'/x],$$

wobei wir im letzten Schritt das syntaktische Substitutionslemma (Lemma 3.22) verwenden. □

Wir definieren eine konkrete Auswertungsstrategie $\to_{par}$ auf der Basis von $\rightrightarrows$. Die Idee ist, dass $s \to_{par} t$ wenn $s \rightrightarrows t$ und t gewissermaßen eine maximale Auffaltung (*englisch: complete development*) von s unter $\rightrightarrows$ ist.

Definition 3.28. Wir definieren $\to_{par}$ induktiv durch die folgenden Klauseln:

1. $s \to_{par} s$, wenn s eine Variable ist;
2. $\lambda x.\, r \to_{par} \lambda x.\, r'$, wenn $r \to_{par} r'$;
3. $pq \to_{par} p'q'$, wenn $p \to_{par} p'$, $q \to_{par} q'$ und p keine λ-Abstraktion ist;
4. $(\lambda x.\, p)q \to_{par} p'[q'/x]$, wenn $p \to_{par} p'$ und $q \to_{par} q'$.

Es ist per Definition offensichtlich, dass $s \to_{par} s'$ stets $s \rightrightarrows s'$ impliziert. Ferner gilt:

Lemma 3.29. *Wenn $s \rightrightarrows s'$ und $s \to_{par} \hat{s}$, dann $s' \rightrightarrows \hat{s}$.*

Beweis. Induktion über die Herleitung von $s \to_{par} \hat{s}$.

1. Wenn s eine Variable ist, dann $s' = \hat{s} = s$, und damit wie oben angemerkt $s' \rightrightarrows \hat{s}$.

2. $s = \lambda x.\, r \to_{par} \lambda x.\, \hat{r} = \hat{s}$ mit $r \to_{par} \hat{r}$. Nach Definition von $\rightrightarrows$ haben wir $s' = \lambda x.\, r'$ und $r \rightrightarrows r'$ für ein geeignetes r'. Per Induktionsvoraussetzung folgt $r' \rightrightarrows \hat{r}$ und daher $s' = \lambda x.\, r' \rightrightarrows \lambda x.\, \hat{r} = \hat{s}$.

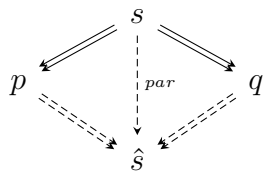
3. $s = pq \to_{par} \hat{p}\hat{q} = \hat{s}$ mit $p \to_{par} \hat{p}$ und $q \to_{par} \hat{q}$, wobei p keine λ-Abstraktion ist. Nach Definition von $\rightrightarrows$ haben wir $s' = p'q'$, $p \rightrightarrows p'$ und $q \rightrightarrows q'$ mit geeigneten p' und q'. Per Induktionsvoraussetzung folgt $p' \rightrightarrows \hat{p}$ und $q' \rightrightarrows \hat{q}$, und damit $s' = p'q' \rightrightarrows \hat{p}\hat{q} = \hat{s}$.

4. $s = (\lambda x.\, p)q \to_{par} \hat{p}[\hat{q}/x] = \hat{s}$, mit $p \to_{par} \hat{p}$, $q \to_{par} \hat{q}$. Es gibt dann zwei Unterfälle.

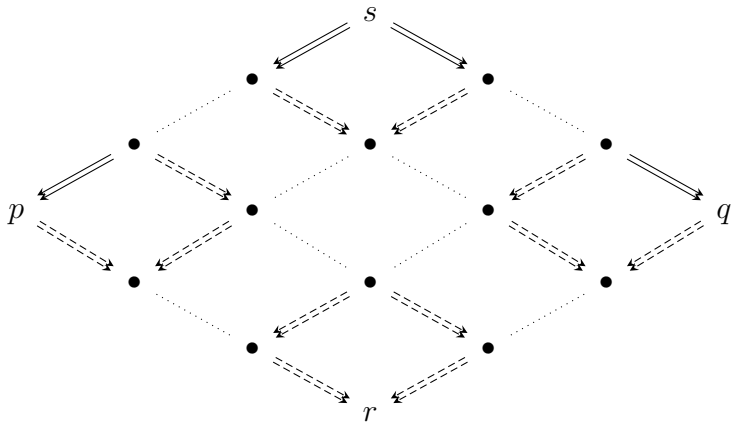
- Der oberste Redex von s wird nicht reduziert. Dann $s' = (\lambda x.\, p')q'$, $p \rightrightarrows p'$ und $q \rightrightarrows q'$ für geeignete p' und q'. Nach Induktionsvoraussetzung folgt $p' \rightrightarrows \hat{p}$, $q' \rightrightarrows \hat{q}$, und daher $s' = (\lambda x.\, p')q' \rightrightarrows \hat{p}[\hat{q}/x] = \hat{s}$.
- Der oberste Redex s wird reduziert. Dann $s' = p'[q'/x]$ für geeignete p' und q', so dass $p \rightrightarrows p'$, $q \rightrightarrows q'$. Nach Induktionsvoraussetzung folgt $p' \rightrightarrows \hat{p}$ und $q' \rightrightarrows \hat{q}$. Nach Lemma 3.27, $p'[q'/x] \rightrightarrows \hat{p}[\hat{q}/x]$, und $s' = p'[q'/x] \rightrightarrows \hat{p}[\hat{q}/x] = \hat{s}$. □

Wir bekommen direkt

Korollar 3.30. *$\rightrightarrows$ hat die Diamant-Eigenschaft, genauer gesagt:*



Der Beweis der Konfluenz von $\to_\beta$ läuft nun wie folgt. Seien $s \to_\beta^\star p$ und $s \to_\beta^\star q$. Nach Lemma 3.26 folgen $s \rightrightarrows^\star p$ und $s \rightrightarrows^\star q$. Durch induktive Anwendung von Korollar 3.30 bekommen wir ein r, so dass $p \rightrightarrows^\star r$ und $q \rightrightarrows^\star r$:



Daraus folgt wiederum nach Lemma 3.26, dass $p \to_\beta^\star r$ und $q \to_\beta^\star r$ □

Im obigen Diagramm ist r von s über $\to_{par}^\star$ erreichbar, und daher hängt r im Endeffekt nur von s ab. Das zeigt übrigens, dass $\to_{par}$ gewissermaßen eine mächtigste Strategie ist, in dem Sinne, dass $\to_{par}$ für jeden per β-Reduktion von s aus erreichbaren Term p stets einen $\to_\beta^\star$-Nachfolger von p erreicht.

Korollar 3.31. *Jeder λ-Term hat höchstens eine NF.*

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

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

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

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

Definition 3.32. Sei $\mathbf{V}$ eine Menge von *Typvariablen* a, b etc. und $\mathbf{B}$ eine Menge von *Basistypen*, etwa $\mathbf{Bool}, \mathbf{Int}$. Die Grammatik für *Typen* $\alpha, \beta, \ldots$ ist dann

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

Bemerkung 3.33. Funktionen mit mehreren Argumenten stellen wir wieder mittels *Currying* dar: Der Typ

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

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

Generell unterscheidet man Typsysteme in zwei Klassen, benannt nach Curry bzw. Church. Bei Typisierung nach Church werden abstrahierte Variablen mit Typen annotiert, in der Form $\lambda x : \alpha\ .t$. Man kann in derartigen Systemen nur typisierbare Terme hinschreiben. Wir arbeiten hier mit Typisierung nach Curry: Wir lassen weiterhin im Prinzip alle Terme in unveränderter Syntax zu und sondern dann die Terme aus, die nach unserem System *typisierbar* sind, d.h. für die ein Typ herleitbar ist. Hierzu muss man wissen, welche Typen die in einem Term vorkommenden freien Variablen haben. Ein *(Typ-)Kontext* ist eine Menge

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

mit paarweise verschiedenen Variablen x_i, denen jeweils ein Typ α_i zugewiesen wird. Wir schreiben Kontexte meist ohne Mengenklammern als $x_1 : \alpha_1; \ldots; x_n : \alpha_n$; insbesondere schreiben wir dann $\Gamma, x : \alpha$ statt $\Gamma \cup \{x : \alpha\}$. Äquivalenterweise verstehen wir einen Kontext Γ wie oben als eine partielle Abbildung von Variablen auf Typen, die auf nur

endlich vielen Variablen definiert ist. Dementsprechend verwenden wir die schon oft verwendete Update-Schreibweise auch für Kontexte: Mit $\Gamma[x \mapsto \alpha]$ bezeichnen wir den Kontext, der x den Typ α zuweist und sich ansonsten wie Γ verhält; in Mengenschreibweise $\Gamma[x \mapsto \alpha] = \{y : \beta \in \Gamma \mid y \neq x\} \cup \{x : \alpha\}$.

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

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

Diese Relation ist induktiv definiert durch die folgenden Regeln:

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

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

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

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:

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

Nun füllen wir diese Lücken von oben nach unten:

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

Beispiel 3.35. Nicht jeder Term ist typisierbar. Beispielsweise ist $\lambda x.\, xx$ (also ω) nicht typisierbar, d.h. es gibt kein α, so dass $\vdash \omega : \alpha$, d.h. so dass ω im leeren Kontext (ω hat ja keine freien Variablen) vom Typ α ist.

Damit ergeben sich die folgenden (Berechnungs)probleme für Typisierung (da $(x_1 : \alpha_1, \ldots, x_n : \alpha_n) \vdash t : \beta \iff \vdash \lambda x_1 \ldots x_n.\, t : \alpha_1 \to \ldots \to \alpha_n \to \beta$, betrachten wir nur Probleminstanzen mit leerem Kontext):

- gilt $\vdash t : \alpha$? (Typüberprüfung, *Type Checking*)
- finde (existiert überhaupt ein?) α mit $\vdash t : \alpha$ (Typinferenz)

- finde (existiert überhaupt ein?) t mit $\vdash t : \alpha$ (dann ist t *inhabited*) (*Type Inhabitation*, automatisches Beweisen, 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; insbesondere gibt es also keinen getypten Fixpunktkombinator.

3.2.1 Elementare Eigenschaften

Lemma 3.37 (Weakening)**.** *Wenn* $\Gamma \vdash t : \alpha$*, dann auch* $\Gamma' \vdash t : \alpha$ *für alle* $\Gamma' \supseteq \Gamma$.

In Worten: Jede Typherleitung klappt auch mit zusätzlichen Annahmen. In Begriffen von Typisierungsregeln ist somit die *Abschwächungsregel*

$$(\text{wk}) \quad \frac{\Gamma \vdash t : \alpha}{\Gamma' \vdash t : \alpha}\ (\Gamma' \supseteq \Gamma)$$

zulässig, d.h. es sind mit ihrer Hilfe keine neuen Typisierungsurteile herleitbar. *Achtung:* Dies bedeutet keineswegs, dass $\Gamma' \vdash t : \alpha$ aus $\Gamma \vdash t : \alpha$ im System ohne (wk) herleitbar wäre!

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

1. Letzte Regel (Ax):

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

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

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

2. Letzte Regel ($\to_e$):

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

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

3. Letzte Regel ($\to_i$):

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

Man prüft leicht, dass $\Gamma[x \mapsto \alpha] \subseteq \Gamma'[x \mapsto \alpha]$. Nach Induktionsvoraussetzung haben wir daher $\Gamma'[x \mapsto \alpha] \vdash t : \beta$, per $(\to_i)$ also $\Gamma' \vdash \lambda x.\, t : \alpha \to \beta$.

□

Lemma 3.38 (Inversionslemma)**.**

1. *Wenn* $\Gamma \vdash x : \alpha$*, dann* $x : \alpha \in \Gamma$*.*
2. *Wenn* $\Gamma \vdash ts : \beta$*, dann existiert* α *mit* $\Gamma \vdash t : \alpha \to \beta$ *und* $\Gamma \vdash s : \alpha$*.*
3. *Wenn* $\Gamma \vdash \lambda x.\, t : \gamma$*, dann hat* γ *die Form* $\gamma = \alpha \to \beta$*, und* $\Gamma[x \mapsto \alpha] \vdash t : \beta$*.*

Beweis. Die Regeln sind syntaxgerichtet. □

3.2.2 Typinferenz

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

Definition 3.39. Wir bezeichnen mit $TV(\alpha)$ die Menge der in einem Typ α vorkommenden Typvariablen, entsprechend $TV(\Gamma)$ für Kontexte Γ. Wir verwenden für Substitutionen von Typvariablen durch Typen dieselben Schreib- und Sprechweisen wir für Substitutionen von Variablen durch Terme, streuen aber gelegentlich zur Betonung den Wortbestandteil „Typ“ ein; z.B. ist eine *Typsubstitution* eben eine Substitution von Typvariablen durch Typen. Eine Typsubstitution σ heißt *Lösung* von $\Gamma \vdash t : \alpha$, wenn $\Gamma\sigma \vdash t : \alpha\sigma$ herleitbar ist, und *allgemeinste Lösung* von $\Gamma \vdash t : \alpha$, wenn σ allgemeinste Typsubstitution unter den Lösungen von $\Gamma \vdash t : \alpha$ ist. Der *Prinzipaltyp von* (Γ, t) (oder von t, wenn Γ leer ist) ist eine (und, wie wir sehen werden, bis auf Variablenumbenennung *die*) allgemeinste Lösung von $\Gamma \vdash t : a$, wobei a „frisch“ ist, d.h. $a \notin TV(\Gamma)$. Das Paar (Γ, t) ist *typisierbar*, wenn eine Lösung von $\Gamma \vdash t : a$ existiert.

Algorithmus 3.40 (Algorithmus W nach Hindley/Milner)**.** Wir definieren rekursiv eine Menge $PT(\Gamma \vdash t : \alpha)$ von Typgleichungen (Notation: $\alpha \doteq \beta$), so dass $PT(\Gamma \vdash t : \alpha)$ genau dann unifizierbar ist, wenn $\Gamma \vdash t : \alpha$ eine Lösung besitzt, und in diesem Fall der allgemeinste Unifikator $\sigma = \mathsf{mgu}(PT(\Gamma \vdash t : \alpha))$ die allgemeinste Lösung von $\Gamma \vdash t : \alpha$ liefert:

1. $PT(\Gamma \vdash x : \alpha) = \{\alpha \doteq \beta \mid x : \beta \in \Gamma\}$

2. $PT(\Gamma \vdash \lambda x.\, t : \alpha) = PT((\Gamma[x \mapsto a]) \vdash t : b) \cup \{a \to b \doteq \alpha\}$, mit a, b frisch.
3. $PT(\Gamma \vdash ts : \alpha) = PT(\Gamma \vdash t : a \to \alpha) \cup PT(\Gamma \vdash s : a)$, mit a frisch.

(Wir verwenden hier einen globalen Begriff von „frisch“, der dafür sorgt, dass in einem Zweig der Rekursion eingeführte frische Variablen nicht nur von im jeweiligen Zweig schon vorhandenen Variablen, sondern auch von allen in anderen Zweigen vorkommenden Variablen verschieden sind; insbesondere sind in der letzten Klausel die in $PT(\Gamma \vdash t : a \to \alpha)$ frisch eingeführten Variablen disjunkt von denen in $PT(\Gamma \vdash s : a)$.) Der Prinzipaltyp von (Γ, t) ist dann die Einschränkung $\mathsf{mgu}(PT(\Gamma \vdash t : a))|_{TV(\Gamma,a)}$ von $\mathsf{mgu}(PT(\Gamma \vdash t : a))$ auf $TV(\Gamma, a)$.

(Der echte Hindley-Milner-Algorithmus arbeitet allerdings mit einer etwas ausdrucksstärkeren Sprache, siehe Abschnitt 5.3.) Wir verwenden dabei den Begriff der Unifikation von Gleichungs*mengen*: eine Substitution ist ein *Unifikator* einer Menge $\mathcal{E}$ von Gleichungen, wenn sie Unifikator aller Gleichungen in $\mathcal{E}$ ist. (Zur Erinnerung: eine Gleichung ist einfach ein Paar von Termen.)

Beispiel 3.41. Wir berechnen den Prinzipaltyp von $(\emptyset, \lambda xy.\, xy)$:

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

Wir erhalten

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

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

Beispiel 3.42. Wir berechnen den Prinzipaltyp von $(x : a, x\lambda z.\, z)$:

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

Wir erhalten

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

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

Beispiel 3.43. Wir vollziehen anhand des Algorithmus nach, dass $\lambda x.\, xx$ nicht typisierbar ist: Wir haben

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

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

Bemerkung 3.44. Es gibt aber stärkere Typsysteme, in denen $\lambda x.xx$ typisierbar ist. Z.B. hat das System $\lambda\cap^-$ einen zusätzlichen Typkonstruktor

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

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

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

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

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

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

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

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

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

Satz 3.45. *Ein Paar (Γ, t) ist genau dann typisierbar, wenn $PT(\Gamma \vdash t : a)$ (mit $a \notin TV(\Gamma)$) unifizierbar ist; in diesem Falle ist $\mathsf{mgu}(PT(\Gamma \vdash t : a))|_{TV(\Gamma,a)}$ ein Prinzipaltyp von (Γ, t).*

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

„$\Leftarrow$“: Jeweils unmittelbar nach Induktionsvoraussetzung und Typregel; z.B. für $t = \lambda x.s$: σ sei erweiterbar zu Unifikator σ' von $PT(\Gamma \vdash \lambda x.s : \alpha) = PT(\Gamma[x \mapsto a] \vdash s :$

$b) \cup \{a \to b \doteq \alpha\}$ für frische a, b. Nach IV (und weil $\Gamma[x \mapsto a]\sigma' = \Gamma\sigma'[x \mapsto \sigma'(a)]$) gilt dann $\Gamma\sigma'[x \mapsto \sigma'(a)] \vdash s : b\sigma'$, also per $(\to_i)$ $\Gamma\sigma' \vdash \lambda x.s : \sigma'(a) \to \sigma'(b) = \alpha\sigma'$ (wobei die Gleichung gerade deswegen gilt, weil σ' unter anderem $a \to b \doteq \alpha$ unifiziert); da $\sigma'|_{TV(\Gamma,\alpha)} = \sigma$, folgt $\Gamma\sigma \vdash t : \alpha\sigma$.

„$\Rightarrow$“:

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

$t = us$: Nach dem Inversionslemma folgt aus $\Gamma\sigma \vdash us : \alpha\sigma$, dass $\Gamma\sigma \vdash u : \gamma \to \alpha\sigma$ und $\Gamma\sigma \vdash s : \gamma$ für ein γ. Sei a eine frische Typvariable und $\sigma' = \sigma[a \mapsto \gamma]$. Dann haben wir $\Gamma\sigma' \vdash u : (a \to \alpha)\sigma'$ und $\Gamma\sigma' \vdash s : a\sigma'$. Nach IV ist also σ' erweiterbar zu einem Unifikator von $PT(\Gamma \vdash u : a \to \alpha)$ und zu einem von $PT(\Gamma \vdash s : a)$; da die in den beiden Typgleichungsmengen jeweils neu hinzukommenden Variablen disjunkt sind, ist σ' damit erweiterbar zu einem Unifikator von $PT(\Gamma \vdash u : a \to \alpha) \cup PT(\Gamma \vdash s : a) = PT(\Gamma \vdash us : \alpha)$.

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

$$\Gamma[x \mapsto a] \vdash s : b,$$

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

3.2.3 Subjektreduktion

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

Lemma 3.46 (Substitutionslemma)**.** $\Gamma[x \mapsto \alpha] \vdash t : \beta$ *und* $\Gamma \vdash s : \alpha \implies \Gamma \vdash t[s/x] : \beta$.

Beweis. Induktion über die Herleitung von $\Gamma[x \mapsto \alpha] \vdash t : \beta$. □

Satz 3.47 (Subjektreduktion)**.** *Wenn* $\Gamma \vdash t : \alpha$ *und* $t \to_\beta^\star s$*, dann auch* $\Gamma \vdash s : \alpha$.

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

Beweis. Wir führen den Beweis für den Fall $t \to_\beta s$; der allgemeine Fall $t \to_\beta^\star s$ folgt dann sofort per Induktion. In diesem Fall haben t und s die Form $t = C((\lambda x.\, u)v)$ und $s = C(u[v/x])$. Wir induzieren über den Kontext $C(\cdot)$.

Falls $C(\cdot) = (\cdot)$, dann $t = (\lambda x.\, u)v$ und $s = u[v/x]$. Nach Inversionslemma folgt aus $\Gamma \vdash (\lambda x.\, u)v : \alpha$, dass $\Gamma \vdash \lambda x.\, u : \beta \to \alpha$ und $\Gamma \vdash v : \beta$ für ein β. Wiederum per Inversionslemma folgt $\Gamma[x \mapsto \beta] \vdash u : \alpha$. Nach dem Substitutionslemma folgt wie verlangt $\Gamma \vdash u[v/x] : \alpha$.

Für einen Kontext der Form $C(\cdot)s$ sei $u \to_\beta u'$, so dass also $C(u)s \to_\beta C(u')s$, und es sei $\Gamma \vdash C(u)s : \alpha$. Nach Inversionslemma existiert dann β mit $\Gamma \vdash C(u) : \beta \to \alpha$ und $\Gamma \vdash s : \beta$. Aus $u \to_\beta u'$ folgt $C(u) \to_\beta C(u')$, also nach Induktionsvoraussetzung $\Gamma \vdash C(u') : \beta \to \alpha$, und dann per $(\to_e)$ $\Gamma \vdash C(u')s : \alpha$. Die beiden verbleibenden Fälle (Kontexte der Form $uC(\cdot)$ und $\lambda x.C(\cdot)$) verlaufen ähnlich. □

3.2.4 Der Curry-Howard-Isomorphismus

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

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

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

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

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

(Achtung: Diese Regeln liefern eine *intuitionistische* Sicht auf Implikation und sind nicht vollständig über der klassischen zweiwertigen Semantik; z.B. ist $((a \to b) \to a) \to a$ klassisch gültig, aber mit obigen Regeln nicht herleitbar.) Im *Sequentenkalkül* werden lokale Annahmen explizit gemacht: ein *Sequent*

$$\Gamma \vdash \phi$$

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

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

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

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

Satz 3.48. *Der Sequent $\vdash \phi$ ist genau dann herleitbar, wenn der Typ ϕ* inhabited *ist.*

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

„⇒“: Wir verstärken die Induktionsbehauptung, indem wir sie auf nichtleere Kontexte verallgemeinern: Zu $\Gamma = \{\phi_1, \ldots, \phi_n\}$ setze $\overline{\Gamma} = \{x_1 : \phi_1, \ldots, x_n : \phi_n\}$. Wir zeigen per Induktion über die Herleitung von $\Gamma \vdash \phi$, dass ein t mit $\overline{\Gamma} \vdash t : \phi$ existiert. Wir unterscheiden wieder über die zuletzt angewendete Regel:

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

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

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

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

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

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

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

In Beweis für „⇒“ betreiben wir *Programmextraktion*: Aus einem Sequentenbeweis von ϕ gewinnen wir ein Programm des Typs ϕ.

3.2.5 Starke Normalisierung für $\lambda\to$

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

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

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

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

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

und setzen dann

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

Definition 3.49. Eine Teilmenge $A \subseteq SN$ heißt *saturiert*, wenn

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

Wir setzen dann

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

Lemma 3.50 (Saturiertheitslemma)**.** $[\![\alpha]\!] \in SAT$ *für alle* α.

Beweis. Induktion über α.

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

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

   (b) Seien $s \in SN$ und $t[s/x]u_1 \dots u_n \in SN$; dann gilt $t, u_1, \dots, u_n \in SN$. Zu zeigen ist $v := (\lambda x.\, t)su_1 \dots u_n \in SN$. Da $t, s, u_1, \dots, u_n \in SN$, hat jede unendliche Reduktionssequenz von v die Form

   $$(\lambda x.\, t)su_1 \dots u_n \to_\beta^\star (\lambda x.\, t')s'u'_1 \dots u'_n \to_\beta t'[s'/x]u'_1 \dots u'_n \to_\beta \dots,$$

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

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

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

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

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

Definition 3.51 (Erfülltheit/Konsequenz)**.** Sei σ eine Substitution. Dann schreiben wir

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

Lemma 3.52 (Korrektheit)**.** *Wenn* $\Gamma \vdash t : \alpha$*, dann* $\Gamma \models t : \alpha$.

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

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

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

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

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

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

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

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

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

letzteres gilt nach Voraussetzung. □

Daraus folgt im wesentlichen sofort unser Zielresultat:

Satz 3.53. *$\lambda\to$ ist stark normalisierend.*

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

4 Induktive und koinduktive Datentypen

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

Beispiel 4.1.

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

- Die obige Datentypdeklaration definiert eine *Signatur* Σ_{Nat}, also eine Menge von *Funktionsymbolen*, die wir als die *Konstruktoren* des Datentyps bezeichnen.
- Die *Semantik* von *Nat* ist definiert als

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

Wir erinnern an dieser Stelle an einen schon in GLoIn gesehenen Begriff:

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

- einem *Grundbereich* (oder Träger oder Universum), d.h. einer Menge M
- für jede n-stellige Funktion f, also $f/n \in \Sigma$, einer Abbildung

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

Im Kontext von induktiven Datentypen bezeichnen wir Σ-Modelle auch als Σ-*Algebren*. Zur Auswertung von Termen mit freien Variablen in einer Σ-Algebra $\mathfrak{M}$ benötigen wir eine *Umgebung*, d.h. eine Abbildung η, die jeder Variablen $v \in V$ einen Wert $\eta(v) \in M$ zuweist. Die Auswertung $\mathfrak{M}[\![t]\!]\eta$ eines Terms t (in der Algebra $\mathfrak{M}$ und unter Umgebung η) ist dann rekursiv definiert durch

$$\begin{aligned} \mathfrak{M}[\![x]\!]\eta &= \eta(x) && (x \in V) \\ \mathfrak{M}[\![f(t_1, \dots, t_n]\!]\eta &= \mathfrak{M}[\![f]\!](\mathfrak{M}[\![t_1]\!]\eta, \dots \mathfrak{M}[\![t_n]\!]\eta) && (f/n \in \Sigma). \end{aligned}$$

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

$$\begin{aligned} [\![Nat]\!][\![0]\!] &= 0 \\ [\![Nat]\!][\![Suc]\!](t) &= Suc(t). \end{aligned}$$

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

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

für alle $g/n \in \Sigma$, $x_1, \ldots, x_n \in M$. Wenn h ferner bijektiv ist, heißt h Σ-*Isomorphismus*. Wenn Σ aus dem Kontext klar ist, wird es in der Terminologie ausgelassen, d.h. wir sprechen dann auch einfach von Homomorphismen und Isomorphismen.

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

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

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

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

Da h injektiv ist, reicht dazu

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

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

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

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

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

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

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

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

Addition von Restklassen ist definiert durch

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

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

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

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

durch

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

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

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

Die nachzuprüfenden Eigenschaften sind hier

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

Dies rechnen wir wie folgt nach:

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

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

Satz 4.10. *Das durch $M = T_\Sigma(\emptyset)$ und*

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

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

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

Wir schreiben $[\![\Sigma]\!]$ für die initiale Σ-Algebra gemäß obigem Satz. Insbesondere ist also z.B. $[\![\Sigma_{Nat}]\!]$ eine initiale Σ_{Nat}-Algebra. Wenn, wie in diesem Fall, die Signatur durch eine Datentypdeklaration gegeben ist, wenden wir diese Schreibweise auch auf den Datentypnamen an, schreiben also z.B. $[\![Nat]\!]$ für die initiale Σ_{Nat}-Algebra.

Beweis (Satz 4.10). Eine Abbildung $h : M \to N$ ist Σ-Homomorphismus $\mathfrak{M} \to \mathfrak{N}$ genau dann, wenn

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

für $f/n \in \Sigma$, $t_1, \ldots, t_n \in M = T_\Sigma(\emptyset)$, d.h. wenn h die rekursiven Gleichungen erfüllt, durch die die Auswertung $\mathfrak{N}[\![-]\!]$ geschlossener Terme in $\mathfrak{N}$ eindeutig definiert ist. □

Satz 4.11. *Die initiale Σ-Algebra ist eindeutig bis auf Isomorphie; genauer: Je zwei initiale Σ-Algebren sind isomorph, und jede zu einer initialen Σ-Algebra isomorphe Σ-Algebra ist selbst wieder initial.*

Beweis. Wir beweisen den zweiten, eher relativ selbstverständlichen, Teil der Aussage zuerst: Es sei $\mathfrak{M}$ eine initiale Σ-Algebra und $f : \mathfrak{M} \to \mathfrak{N}$ ein Σ-Isomorphismus. Zu zeigen ist, dass dann auch $\mathfrak{N}$ initiale Σ-Algebra ist. Für eine gegebene Σ-Algebra $\mathfrak{P}$ haben wir einen eindeutigen Σ-Homomorphismus $h : \mathfrak{M} \to \mathfrak{P}$ und damit einen Σ-Homomorphismus $h \circ f^{-1} : \mathfrak{N} \to \mathfrak{P}$. Zur Eindeutigkeit: Ein gegebener Σ-Homomorphismus $k \colon \mathfrak{N} \to \mathfrak{P}$ liefert einen Σ-Homomorphismus $k \circ f \colon \mathfrak{M} \to \mathfrak{P}$. Wegen der Eindeutigkeit von h gilt $h = k \circ f$, woraus $k = h \circ f^{-1}$ folgt.

Es bleibt der erste Teil der Aussage: Seien $\mathfrak{M}, \mathfrak{N}$ initiale Σ-Algebren. Dann haben wir Σ-Homomorphismen

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

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

Hierauf begründen wir die allgemeine Definition der Semantik von Datentypen: Die oben angedeutete Syntax data D where ... für einen Datentyp D deklariert eine Signatur Σ_D und definiert, im wesentlichen eindeutig, die initiale Σ_D-Algebra $[\![D]\!] = [\![\Sigma_D]\!]$, repräsentiert als $T_{\Sigma_D}(\emptyset)$ mit der wie oben definierten Σ_D-Algebrastruktur.

Definition 4.12 (Notation zu Mengenkonstruktionen)**.** Sei I eine Indexmenge und $(X_i)_{i\in I}$ eine über I indizierte Familie von Mengen (man stelle sich diese als „Typen“ vor). Dann schreiben wir (wie üblich)

$$\begin{aligned} \textstyle\prod_{i\in I} X_i &= \{(x_i)_{i\in I} \mid x_i \in X_i \text{ für alle } i \in I\} && (\text{„struct“}) \\ \textstyle\sum_{i\in I} X_i &= \{(i, x) \mid i \in I \text{ und } x \in X_i\} && (\text{„union“}) \end{aligned}$$

Seien nun $f_i : X_i \to Y_i$, $g_i : X_i \to Z$ und $h_i : Z \to X_i$ für $i \in I$. Dann haben wir Abbildungen

$$\begin{aligned} \textstyle\prod_{i\in I} f_i &: \textstyle\prod_{i\in I} X_i \to \prod_{i\in I} Y_i, & (\textstyle\prod_{i\in I} f_i)((x_i)_{i\in I}) &= (f_i(x_i))_{i\in I} \\ \textstyle\sum_{i\in I} f_i &: \textstyle\sum_{i\in I} X_i \to \sum_{i\in I} Y_i, & (\textstyle\sum_{i\in I} f_i)(j, x) &= (j, f_j(x)) \\ [g_i]_{i\in I} &: \textstyle\sum_{i\in I} X_i \to Z, & [g_i]_{i\in I}(j, x) &= g_j(x) \\ \langle h_i\rangle_{i\in I} &: Z \to \textstyle\prod_{i\in I} X_i, & \langle h_i\rangle_{i\in I}(z) &= (h_i(z))_{i\in I} \\ in_j &: X_j \to \textstyle\sum_{i\in I} X_i, & in_j(x) &= (j, x) \\ \pi_j &: \textstyle\prod_{i\in I} X_i \to X_j, & \pi_j((x_i)_{i\in I}) &= x_j. \end{aligned}$$

Den Indexbereich '$i \in I$' lassen wir in der Notation of weg und schreiben z.B. nur $\sum X_i$. Für I endlich, sagen wir $I = \{1, \ldots, n\}$, schreiben wir endliche Summen und Produkte wie $X_1 + X_2 + X_3$ oder $f_1 \times f_2$, und listen Abbildungen zwischen $[\ldots]$ oder $\langle \ldots \rangle$ einfach auf, z.B. $[g_1, g_2, g_3]$. Wenn in Produkten alle 'Faktoren' gleich sind, verwenden wir Potenzschreibweise, z.B. $X^2 = X \times X$ und $f^3 = f \times f \times f$. Für X^0 und f^0 schreiben wir 1; als Menge ist also $1 = \{()\}$, und als Abbildung die identische Abbildung auf dieser Menge.

Lemma 4.13. *In Bezeichnungen wie oben gilt*

- $[g_i] \circ in_j = g_j$
- $\sum f_i = [in_i \circ f_i]$
- $[r \circ in_i] = r$ *für* $r \colon \left(\sum X_i\right) \to Z$.
- $\pi_j \circ \langle h_i \rangle = h_j$
- $\prod f_i = \langle f_i \circ \pi_i \rangle$
- $\langle \pi_i \circ f \rangle = f$ *für* $f : Z \to \prod X_i$

Insbesondere ist jede Abbildung $Z \to \prod X_i$ von der Form $\langle h_i \rangle$, und jede Abbildung $\sum X_i \to Z$ von der Form $[g_i]$.

Beweis. Wir haben

$$
\begin{aligned}
&[g_i](in_j(x)) = [g_i](j, x) = g_j(x)\\
&[in_i \circ f_i](j, x) = in_j(f_j(x)) = (j, f_j(x)) = (\textstyle\sum f_i)(j, x)\\
&[r \circ in_i](j, x) = r(in_j(x)) = r(j, x)\\
&\pi_j \circ \langle h_i \rangle(z) = \pi_j((h_i(z))_{i \in I}) = h_j(z)\\
&\langle f_i \circ \pi_1 \rangle((x_k)_{k \in I}) = (f_i(\pi_i((x_k)_{k \in I})))_{i \in I} =\\
&\qquad\qquad (f_i(x_i))_{i \in I} = (\textstyle\prod f_i)((x_i)_{i \in I}\\
&\langle \pi_i \circ f \rangle_{i \in I}(z) = (\pi_i(f(z))_{i \in I} = f(z),
\end{aligned}
$$

wobei man die letzte Gleichheit durch komponentenweisen Vergleich sieht. □

Wir sehen damit, dass Σ-Algebren $\mathfrak{M}$ eins zu eins Abbildungen der Form

$$\alpha \colon \textstyle\sum_{f/n \in \Sigma} M^n \to M$$

entsprechen: Wir erhalten α aus $\mathfrak{M}$ als

$$\alpha = [\mathfrak{M}[\![f]\!]]_{f \in \Sigma},$$

und umgekehrt erhalten wir $\mathfrak{M}$ aus α per

$$\mathfrak{M}[\![f]\!] = \alpha \circ in_f;$$

nach obigem Lemma sind diese beiden Konstruktionen zueinander invers. Explizit korrespondieren α und $\mathfrak{M}$ damit per

$$\alpha(f, (x_1, \ldots, x_n)) = \mathfrak{M}[\![f]\!](x_1, \ldots, x_n) \tag{10}$$

für $f/n \in \Sigma$. Wir schreiben kurz F_Σ für die durch Σ wie oben induzierte Konstruktion auf Mengen und Abbildungen, also

$$F_\Sigma M = \sum_{f/n \in \Sigma} M^n \qquad F_\Sigma h = \sum_{f/n \in \Sigma} h^n.$$

Damit ist dann eine Σ-Algebra mit Grundbereich M einfach eine Abbildung $\alpha\colon F_\Sigma M \to M$. Auch die Homomorphieeigenschaft können wir in diesen Begriffen umformulieren:

Lemma 4.14. *Seien $\mathfrak{M}$, $\mathfrak{N}$ Σ-Algebren, denen Abbildungen $\alpha\colon F_\Sigma M \to M$ bzw. $\beta\colon F_\Sigma N \to N$ entsprechen. Eine Abbildung $h : M \to N$ ist genau dann ein Homomorphismus $\mathfrak{M} \to \mathfrak{N}$, wenn das Diagramm*

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

kommutiert.

Beweis. Für $f/n \in \Sigma$, $x_1, \ldots, x_n \in M$ lautet die Homomorphiebedingung

$$h(\mathfrak{M}[\![f]\!](x_1, \ldots, x_n) = \mathfrak{N}[\![f]\!](h(x_1), \ldots, h(x_n)). \tag{11}$$

Nun gilt

$$h(\alpha(f, (x_1, \ldots, x_n))) = h(\mathfrak{M}[\![f]\!](x_1, \ldots, x_n)) \tag{12}$$

per Definition von α gemäß (10), und ferner

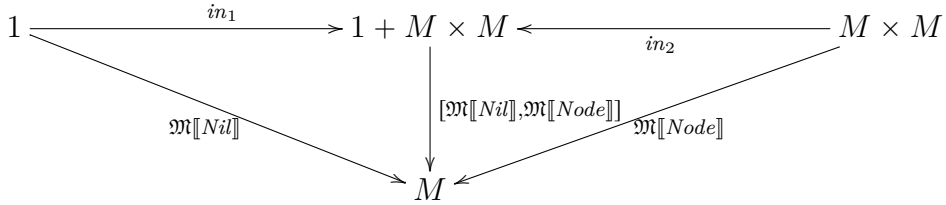
$$\begin{aligned} \beta((F_\Sigma h)(f, (x_1, \ldots, x_n))) &= \beta(f, (h(x_1), \ldots, h(x_n))) \\ &= \mathfrak{N}[\![f]\!](h(x_1), \ldots, h(x_n)) \end{aligned}$$

per Definition von $F_\Sigma h$ und β. Damit ist also (11) äquivalent zu $h(\alpha(f, (x_1, \ldots, x_n))) = \beta((F_\Sigma h)(f, (x_1, \ldots, x_n)))$. □

Beispiel 4.15. Wir illustrieren die obigen Konstruktionen anhand der Datentypdeklaration

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

(wir führen das Beispiel hier im Skript deutlich detaillierter durch als in der Vorlesung). Eine Σ_{tree}-Algebra $\mathfrak{M}$ lässt sich dann darstellen als eine Abbildung $\alpha = [\mathfrak{M}[\![Nil]\!], \mathfrak{M}[\![Node]\!]] : 1 + M \times M \to M$, und umgekehrt induziert jede Abbildung $\alpha : 1 + M \times M \to M$ eine Σ_{tree}-Algebra per $\mathfrak{M}[\![Nil]\!] = \alpha(in_1(*))$ und $\mathfrak{M}[\![Node]\!](x, y) = \alpha(in_2(x, y))$:



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

$$\begin{array}{ccc} M \times M & \xrightarrow{h \times h} & N \times N \\ {\scriptstyle \mathfrak{M}[\![Node]\!]}\downarrow & & \downarrow{\scriptstyle \mathfrak{N}[\![Node]\!]} \\ M & \xrightarrow[h]{} & N \end{array} \tag{13}$$

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

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

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

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

$$\begin{array}{ccc} 1 + M \times M & \xrightarrow{1+h\times h} & 1 + N \times N \\ {\scriptstyle [\mathfrak{M}[\![Nil]\!], \mathfrak{M}[\![Node]\!]]}\downarrow & & \downarrow{\scriptstyle [\mathfrak{N}[\![Nil]\!], \mathfrak{M}[\![Node]\!]]} \\ M & \xrightarrow[h]{} & N \end{array} \tag{14}$$

kommutiert – z.B. erhalten wir aus diesem Diagramm das Diagramm (13) durch Postkomponieren mit in_2:

$$\begin{aligned} h \circ \mathfrak{M}[\![Node]\!] &= h \circ [\mathfrak{M}[\![Nil]\!], \mathfrak{M}[\![Node]\!]] \circ in_2 && \text{(Lemma 4.13)} \\ &= [\mathfrak{N}[\![Nil]\!], \mathfrak{M}[\![Node]\!]] \circ (1 + h \times h) \circ in_2 && \text{(14)} \\ &= [\mathfrak{N}[\![Nil]\!], \mathfrak{M}[\![Node]\!]] \circ [\ldots, in_2 \circ (h \times h)] \circ in_2 && \text{(Lemma 4.13)} \\ &= [\mathfrak{N}[\![Nil]\!], \mathfrak{M}[\![Node]\!]] \circ in_2 \circ (h \times h) && \text{(Lemma 4.13)} \\ &= \mathfrak{M}[\![Node]\!] \circ (h \times h) && \text{(Lemma 4.13).} \end{aligned}$$

4.1 Initialität und Rekursion

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

$$h(f(x_1, \ldots, x_n)) = a_f(h(x_1), \ldots, h(x_n)), \tag{15}$$

wobei wir für jedes $f/n \in \Sigma$ eine Funktion $a_f : A^n \to A$ wählen. Die Wahl der a_f definiert nämlich eine Σ-Algebra mit Träger A, und Gleichung (15) besagt einfach, dass h ein Homomorphismus nach A ist.

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

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

Z.B. liefert

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

eine Funktion, die die Anzahl Blätter eines Baums zählt.

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

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

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

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

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

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

$$\begin{aligned} add\ 0\ k &= k \\ add\ (Suc\ n)\ k &= Suc\ (add\ m\ k). \end{aligned}$$

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

$$add\colon Nat \to (Nat \to Nat).$$

Wir können dann beispielsweise die obige Definition umschreiben zu

$$\begin{aligned} add\ 0 &= \lambda k.\, k \\ add\ (Suc\ n) &= \lambda k.\, Suc\ (add\ n\ k). \end{aligned}$$

Zusammenfassend haben wir

Definition 4.16. Eine *primitiv rekursive Definition* einer Funktion h besteht aus je einer Gleichung der Form

$$\begin{aligned} h(f(x_1, \ldots, x_n), y_2, \ldots, y_k) = g_f(&h(x_1, t_{12}, \ldots, t_{1k}), \ldots, \\ &h(x_n, t_{n2}, \ldots, t_{nk}), x_1, \ldots, x_n, y_2, \ldots, y_k) \end{aligned}$$

für jeden Konstruktor $f/n \in \Sigma$, wobei die t_{ij} Terme über den Variablen $x_1, \ldots, x_n, y_2, \ldots, y_k$ sind. Dabei nennen wir den Ausdruck $f(x_1, \ldots, x_n)$ ein *Konstruktor-Pattern*.

(Man beachte also, dass die rekursiven Aufrufe von h in den nicht-rekursiven Positionen auch andere Argumente als die y_i verwenden können.)

Satz 4.17. *Eine primitiv rekursive Definition von h definiert h eindeutig.*

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

4.2 Mehrsortigkeit

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

Beispiel 4.18.

```
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
6         -- oder Cons: Tree a -> Forest a -> Forest a
```

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

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

wobei wir a als *Parametersorte* markieren; wir erledigen hier im gleichen Zuge das Problem, dass unsere Datenstrukturen bisher gewissermaßen nur Struktur, aber keinen Inhalt hatten – Parametersorten denken wir uns gerade als die Typen von *Einträgen* in unseren Datenstrukturen, im obigen Beispiel an den Blättern des Baums.

Im allgemeinen stellt sich dies wie folgt dar:

Definition 4.19. Eine *sortierte Signatur* $\Sigma = (S_0, S, F)$ besteht aus:

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

(Die Idee hinter der Einschränkung $b \in S\backslash S_0$ ist, dass die Parameter in S_0 als vorgegebene Typen angesehen werden, für die wir keine neuen Werte konstruieren wollen.) Wir weisen nun, ähnlich wie im getypten λ-Kalkül, Termen Sorten zu:

Definition 4.20.

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

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

$$\frac{\Gamma \vdash t_1 : a_1 \quad \ldots \quad \Gamma \vdash t_n : a_n}{\Gamma \vdash c(t_1, \ldots, t_n) : b}\ (c : a_1 \times \cdots \times a_n \to b)$$

  Wir schreiben

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

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

Entsprechend haben wir auch mehrsortige semantische Begriffe:

Definition 4.21. Sei $V = (V_a)_{a \in S_0}$ eine Mengenfamilie. Eine (mehrsortige) Σ-*Algebra* $\mathfrak{M}$ *über* V besteht aus

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

- einer Abbildung
$$\mathfrak{M}[\![c]\!] : \mathfrak{M}[\![a_1]\!] \times \cdots \times \mathfrak{M}[\![a_n]\!] \to \mathfrak{M}[\![b]\!]$$
für jedes $c : a_1 \times \cdots \times a_n \to b$ in Σ.

Gegeben eine Σ-Algebra $\mathfrak{M}$ und eine *Umgebung* η, die jeder Variablen $x : a \in \Gamma$ einen Wert $\eta(x) \in \mathfrak{M}[\![a]\!]$ zuordnet, definieren wir *Termauswertung* für Terme im Kontext Γ im wesentlichen wie bisher: Für einen Term $\Gamma \vdash t : b$ definieren wir einen Wert

$$\mathfrak{M}[\![t]\!]\eta \in \mathfrak{M}[\![b]\!]$$

rekursiv per

$$\mathfrak{M}[\![x]\!] = \eta(x) \qquad \mathfrak{M}[\![c(t_1, \ldots, t_n)]\!] = \mathfrak{M}[\![c]\!](\mathfrak{M}[\![t_1]\!]\eta, \ldots, \mathfrak{M}[\![t_n]\!]\eta).$$

Ein Σ-*Homomorphismus* $g : \mathfrak{M} \to \mathfrak{N}$ zwischen Σ-Algebren $\mathfrak{M}, \mathfrak{N}$ über V ist eine Familie $g = (g_a)_{a \in S}$ von Abbildungen
$$g_a : \mathfrak{M}[\![a]\!] \to \mathfrak{N}[\![a]\!]$$
mit $g_a = id$ für $a \in S_0$ und

$$g_b(\mathfrak{M}[\![c]\!](x_1, \ldots, x_n)) = \mathfrak{N}[\![c]\!](g_{a_1}(x_1), \ldots, g_{a_n}(x_n))$$

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

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

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

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

Satz 4.22 (Initialität/Folding)**.** *Für alle Σ-Algebren $\mathfrak{N}$ über V existiert genau ein Σ-Homomorphismus $g : [\![\Sigma]\!]_V \to \mathfrak{N}$ über V.*

Beweis. Die Anforderungen an g, instanziiert für die Definition von $[\![\Sigma]\!]$ als Σ-Algebra über V, lauten

$$\begin{aligned} g(x) &= x \\ g(c(t_1, \ldots, t_n)) &= \mathfrak{N}[\![c]\!](g(t_1), \ldots, g(t_n)) \end{aligned}$$

und entsprechen damit genau der Definition von Termauswertung $\mathfrak{N}[\![-]\!]\eta$ über der durch $\eta(x) = x$ definierten Umgebung η. □

Damit haben wir also wieder eine *fold*-Operation, die als Argumente die Komponenten einer (jetzt mehrsortigen) Algebra nimmt und jetzt *mehrere* gegenseitig rekursiv definierte Funktionen liefert, nämlich eine auf jeder Nicht-Parametersorte des Datentyps (siehe auch das folgende Beispiel). Ganz analog wie im einsortigen Fall bekommen wir primitive Rekursion als allgemeineres Definitionsprinzip; auch hier werden dann wieder mehrere Funktionen gegenseitig rekursiv definiert, eine pro Nicht-Parametersorte des Datentyps.

Beispiel 4.23. Sei Σ_{Tree} die im Eingangsbeispiel deklarierte mehrsortige Signatur. Wir definieren eine Σ_{Tree}-Algebra über $V_a = \mathbb{N}$ per

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

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

4.3 Strukturelle Induktion auf Datentypen

Als Beweisprinzip für rekursive Funktionen bietet sich typischerweise Induktion an; dabei sollte das verwendete Induktionsprinzip dieselbe Struktur haben wie die rekursive Definition. Als Beispiel diene zunächst folgende rekursive Definition von Konkatenation:

Beispiel 4.24.

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

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

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

beweisen wollen, induzieren wir über l, wie folgt:

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

Cons x l: Wir haben

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

sowie

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

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

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

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

4.4 Induktion über mehrsortige Datentypen

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

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

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

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

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

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

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

Die letzteren Funktionen lassen sich auch direkt als *fold* schreiben:

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

(Das geht auch für $(\mathit{mirrort}, \mathit{mirrorf})$; wie?) Wir wollen nun

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

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

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

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

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

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

Leaf x: Einerseits haben wir

$$\mathit{flattent}\ (\mathit{mirrort}\ (\mathit{Leaf}\ x)) = \mathit{flattent}\ (\mathit{Leaf}\ x) = [x].$$

Andererseits gilt

$$\mathit{rev}\ (\mathit{flattent}\ (\mathit{Leaf}\ x)) = \mathit{rev}\ [x],$$

und man errechnet leicht $\mathit{rev}\ [x] = [x]$.

Nil. Es gilt

$$\mathit{flattenf}\ (\mathit{mirrorf}\ \mathit{Nil}) = \mathit{flattenf}\ \mathit{Nil} = \mathit{Nil}$$

und

$$\mathit{rev}\ (\mathit{flattenf}\ \mathit{Nil}) = \mathit{rev}\ \mathit{Nil} = \mathit{Nil}.$$

Node f: Wir haben einerseits

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

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

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

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

Andererseits haben wir auch

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

Es verbleiben unsere beiden Hilfsaussagen:

Lemma 4.25 (Lemma A)**.** *Es gilt* $\mathit{flattenf}\ (\mathit{concat}\ f\ g) = \mathit{concat}\ (\mathit{flattenf}\ f)\ (\mathit{flattenf}\ g)$

Beweis. Induktion über f:

Nil: Wir haben

$$\mathit{flattenf}\ (\mathit{concat}\ \mathit{Nil}\ g) = \mathit{flattenf}\ g$$

und

$$\mathit{concat}\ (\mathit{flattenf}\ \mathit{Nil})\ (\mathit{flattenf}\ g) = \mathit{concat}\ \mathit{Nil}\ (\mathit{flattenf}\ g) = \mathit{flattenf}\ g.$$

Cons t f: Wir haben

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

sowie

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

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

Lemma 4.26 (Lemma B)**.** *Es gilt flattenf* $[t] =$ *flattent* t*.*

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

4.5 Kodatentypen

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

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

Definition 4.27 (Streams)**.** Ein *Stream* über einem Alphabet A ist eine unendliche Sequenz

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

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

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

und

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

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

Notation 4.28. Wir verwenden eine zu unserer Syntax für Datentypen duale Syntax für Codatentypen, in der wir einfach die Destruktoren eines Kodatentyps auflisten. Streams deklariert man dann in der Form

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$Gf : GX \to GY.$$

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

Bemerkung 4.29. Wir erinnern daran, dass wir zu einer (Datentyp-)Signatur Σ den Mengenoperator

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

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

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

Dual dazu haben wir

Definition 4.30. Eine *G-Koalgebra* für einen Mengenoperator G ist eine Abbildung $M \to GM$. Die Elemente von M nennen wir *Zustände*.

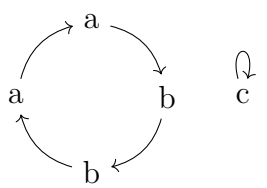
Beispiel 4.31 (Streams)**.** Wenn wir annehmen, dass die Parametersorte a durch die Menge A interpretiert wird, gehört zur Kodatentypdeklaration *Stream a* der Mengenoperator

$$GX = A \times X.$$

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

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

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



eine G-Koalgebra dar.

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

Wir erinnern daran, dass in Begriffen von Mengenoperatoren eine Abbildung $f : \mathfrak{M} \to \mathfrak{N}$ genau dann ein Σ-Homomorphismus zwischen Σ-Algebren $\mathfrak{M}$, $\mathfrak{N}$ ist, wenn folgendes Diagramm kommutiert:

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

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

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

kommutiert.

Beispiel 4.32 (Streams)**.** Für $GX = A \times X$ ist $f : \mathfrak{M} \to \mathfrak{N}$ ein G-Koalgebramorphismus zwischen G-Koalgebren $\mathfrak{M}$, $\mathfrak{N}$ genau dann, wenn

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

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

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

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

Dual zum Begriff der initialen Algebra haben wir:

Definition 4.33. Eine G-Koalgebra $\mathfrak{N}$ ist *final*, wenn für jede G-Koalgebra $\mathfrak{M}$ genau ein G-Koalgebramorphismus $\mathfrak{M} \to \mathfrak{N}$ existiert.

Satz 4.34. *Finale G-Koalgebren sind eindeutig bis auf Isomorphie.*

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

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

Satz 4.35. *Sei $GX = A \times X$. Die G-Koalgebra $\langle hd, tl\rangle : A^\omega \to A \times A^\omega$ ist final.*

(Wir bezeichnen im folgenden die finale G-Koalgebra kurz mit A^ω.)

Beweis. Sei $\mathfrak{N}$ eine G-Koalgebra; wir schreiben kurz $\mathfrak{N}[\![hd]\!] = hd_\mathfrak{N}$, $\mathfrak{N}[\![tl]\!] = tl_\mathfrak{N}$. Für einen Zustand $x \in N$ definieren wir $f : N \to A^\omega$ als $f(x) = (a_0, a_1, \dots)$ mit

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

Zu zeigen sind:

1. *f ist G-Koalgebramorphismus $\mathfrak{N} \to A^\omega$:* Wir haben

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

   und

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

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

   $i = 0$: Wir haben

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

   $i \to i + 1$: Wir haben

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

□

Definition 4.36 (Unfold)**.** Für $h : M \to A$, $t : M \to M$ wird per Finalität eine Funktion $\mathit{unfold}\ h\ t : M \to A^\omega$ definiert durch

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

Beispiel 4.37. Wir können einen Stream *ones*, also eine Abbildung $\mathit{ones} : 1 \to \mathbb{N}^\omega$, corekursiv definieren durch

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

oder in einer Zeile per $\mathit{ones} = \mathit{unfold}\ (\lambda x.\, 1)\ \mathit{id}_1$.

Beispiel 4.38. Wir wollen eine Funktion $\mathit{blink} : a \times \mathit{Stream}\ a \to \mathit{Stream}\ a$ definieren, so dass $\mathit{blink}\ x\ (a_0, a_1, \dots) = (x, a_0, x, a_1, x, a_2, \dots)$:

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

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

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

wobei 2 der Datentyp mit konstanten Konstruktoren 0, 1 ist. Da wir Elemente von Koalgebren als Zustände verstehen, erweitern wir also intuitiv gesprochen den Speicher um ein Bit, das anzeigt, ob wir an gerader oder ungerader Position im Stream sind. Wir wollen das Verhalten

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

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

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

Bemerkung 4.39. Wie das Stream-Beispiel illustriert, können wir die Elemente der finalen G-Koalgebra als die möglichen Verhalten von Zuständen in G-Koalgebren ansehen. In diesem Sinne bildet der eindeutige Morphismus von einer G-Koalgebra in die finale G-Koalgebra einen Zustand auf sein *Verhalten* ab. In diesen Begriffen haben wir:

1. Jeder Zustand in der finalen Koalgebra ist sein eigenes Verhalten (denn: die Identität ist der eindeutige Morphismus der finalen Koalgebra in sich selbst).
2. Morphismen von G-Koalgebren bewahren Verhalten (denn: wenn $f : \mathfrak{N}_1 \to \mathfrak{N}_2$ ein Morphismus von G-Koalgebren ist, $\mathfrak{M}$ die finale G-Koalgebra, und $g : \mathfrak{N}_2 \to \mathfrak{M}$, dann ist $g \circ f$ der eindeutige Morphismus $\mathfrak{N}_1 \to \mathfrak{M}$, d.h. das Verhalten von $x \in N_1$ ist das gleiche wie das von $f(x) \in N_2$, nämlich $g(f(x))$).

4.6 Koinduktion

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

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

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

$$\mathit{odd}\ (\mathit{blink}\ 0\ x\ s) = s$$

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

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

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

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

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

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

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

Definition 4.40. Eine Relation $R \subseteq A^\omega \times A^\omega$ heißt *Bisimulation*, wenn für alle $(s,t) \in R$ gilt:

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

Satz 4.41 (Koinduktion)**.** *Wenn R eine Bisimulation ist, dann gilt $R \subseteq id$, d.h.*

$$sRt \implies s = t$$

für all $s, t \in A^\omega$.

Beweis. Sei sRt. Per Induktion über n folgt

$$tl^i(s)\, R\, tl^i(t) \qquad \text{für alle } i \geqslant 0,$$

und damit

$$hd(tl^i(s)) = hd(tl^i(t)) \qquad \text{für alle } i \geqslant 0.$$

Damit folgt $s = t$, da ja $hd \circ tl^i$ gerade das i-te Element eines Streams berechnet. □

Bemerkung 4.42. Man kann obigen Satz als *Korrektheit* des Bisimulationsprinzips auffassen: Wenn ich zwei Streams durch eine Bisimulation in Beziehung setzen kann, sind sie tatsächlich gleich. Es gilt in dieser Formulierung trivialerweise auch *Vollständigkeit*, also die Umkehrung:: Wenn $s = t$, dann existiert eine Bisimulation R mit sRt, nämlich $R = id$. (Man überzeugt sich leicht, dass id in der Tat eine Bisimulation ist.)

Beispiel 4.43. Mit Hilfe des Bisimulationsprinzips führen wir jetzt den Beweis, dass in der Tat $odd\ (blink\ 0\ x\ s) = s$ gilt. Dazu müssen wir eine geignete Bisimulation erfinden. Wir versuchen es mit

$$R = \{(odd\ (b_0\ x\ s), s) \mid x \in A, s \in A^\omega\},$$

wobei wir wieder $blink\ i$ als b_i abkürzen. Wir rechnen nach, dass R in der Tat eine Bisimulation ist:

1. $hd\ (odd\ (b_0\ x\ s)) = hd\ (tl\ (b_0\ x\ s)) = hd\ (b_1\ x\ s) = hd\ s$
2. $tl\ (odd\ (b_0\ x\ s)) = odd\ (tl\ (tl\ (b_0\ x\ s))) = odd\ (tl\ (b_1\ x\ s)) = odd\ (b_0\ x\ (tl\ s))\ R\ tl\ s$

also gilt $odd\ (b_0\ x\ s) = s$ für alle x, s.

Beispiel 4.44. Die Funktion *merge* mischt zwei Streams durch abwechselndes Auflisten der jeweiligen Elemente, d.h.

$$merge\ (x_0, \dots)\ (y_0, \dots) = (x_0, y_0, x_1, y_1, \dots).$$

Diese Funktion können wir korekursiv definieren durch

$$\begin{aligned} hd\ (merge\ s\ t) &= hd\ s \\ tl\ (merge\ s\ t) &= merge\ t\ (tl\ s) \end{aligned}$$

Wir wollen nun zeigen, dass

$$merge\ (even\ s)\ (odd\ s) = s.$$

Wir versuchen dazu zunächst zu zeigen, dass

$$R = \{(merge\ (even\ s)\ (odd\ s), s) \mid s \in A^\omega\}$$

eine Bisimulation ist. Wir rechnen wie folgt:

1. $hd\ (merge\ (even\ s)\ (odd\ s)) = hd\ (even\ s) = hd\ s$
2. $tl\ (merge\ (even\ s)\ (odd\ s)) = merge\ (odd\ s)\ (tl\ (even\ s)) = merge\ (odd\ s)\ (even\ (tl\ (tl\ s)))$. Es ist nicht klar, dass letzterer Term in Relation R zu $tl\ s$ steht.

Die Relation R scheint insofern zu klein gewählt. Wir setzen daher

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

– dies ist analog zur Verstärkung der Induktionsbehauptung im induktiven Fall. Wir rechnen nach, dass R' nunmehr in der Tat eine Bisimulation ist: Die nachzurechnenden Gleichungen für Paare in R haben wir oben bereits geprüft, wobei das Fehlschlagen der Bedingung für tl durch die Erweiterung auf R' jetzt behoben ist. Für die neuen Paare in R' rechnen wir wie folgt:

1. $hd\ (merge\ (odd\ s)\ (\dots)) = hd\ (odd\ s) = hd\ (tl\ s)$
2. $tl\ (merge\ (odd\ s)\ (even\ (tl\ (tl\ s)))) = merge\ (even\ (tl\ (tl\ s)))\ (tl\ (odd\ s))$
   $= merge\ (even\ (tl\ (tl\ s)))\ (odd\ (tl\ (tl\ s)))\quad R'\quad tl\ (tl\ s)$

4.7 Kodatentypen mit mehreren Nachfolgeroperationen

[Dieser Abschnitt wird bei Zeitknappheit statt des folgenden Abschnitts „Kodatentypen mit Alternativen“ verwendet.]

Wir machen uns kurz klar, wie die im vorigen Abschnitt eingeführten Konzepte für Kodatentypen K funktionieren, die mehr als einen Observer des Typs $K \to K$ haben, also mehrere Nachfolgeroperatoren. Unser führendes Beispiel ist der folgende Kodatentyp unendlicher binärer Bäume:

```
1 codata Tree where
2   out: Tree -> A
3   left: Tree -> Tree
4   right: Tree -> Tree
```

Die durch diesen Kodatentyp deklarierte Mengenkonstruktion ist

$$G = A \times id \times id$$

(bei etwas saloppem Umgang mit dem mehrfachen kartesischen Produkt, das wir insbesondere ungeklammert schreiben). Die finale G-Koalgebra besteht in der Tat aus unendlichen binären Bäumen, in denen jeder Knoten mit einem Element aus A beschriftet sind und jeweils einen rechten und einen linken Nachfolger hat. Die Finalität zeigt man ganz analog wie im Falle der Streams, unter Ersetzung der Induktion über natürliche Zahlen durch Induktion über Adressen von Knoten im Baum (wie würden Sie solche Adressen definieren?). Dies führt zu einem entsprechenden Begriff von korekursiver Definition, in der weiterhin für jeden Beobachter (im Beispiel drei) eine Klausel anzugeben ist; z.B. definieren wir eine Funktion $\mathit{mirror} : \mathit{Tree} \to \mathit{Tree}$ (wir identifizieren zur besseren Lesbarkeit ab jetzt gelegentlich syntaktische Namen mit ihrer semantischen Interpretation, hier den Kodatentyp *Tree* mit der Menge der unendlichen binären Bäume) korekursiv durch

$$\begin{aligned} \mathit{out}\ (\mathit{mirror}\ t) &= \mathit{out}\ t \\ \mathit{left}\ (\mathit{mirror}\ t) &= \mathit{mirror}\ (\mathit{right}\ t) \\ \mathit{right}\ (\mathit{mirror}\ t) &= \mathit{mirror}\ (\mathit{left}\ t). \end{aligned}$$

Der Begriff der Bisimulation verallgemeinert sich ebenfalls zwanglos, indem man Stabilität der Bisimulation unter allen Nachfolgeroperationen (statt wie im Falle von Streams nur unter einer) fordert; z.B. ist eine Relation R auf der finalen G-Koalgebra mit G wie oben eine Bisimulation, wenn aus sRt stets folgt, dass

- $\mathit{out}(s) = \mathit{out}(t)$;
- $\mathit{left}(s)\ R\ \mathit{left}(t)$; und
- $\mathit{right}(s)\ R\ \mathit{right}(t)$.

Man zeigt dann wie zuvor, dass aus sRt stets $s = t$ folgt. Wir verwenden dieses Prinzip, um die erwartete Gleichung

$$\mathit{mirror}\ (\mathit{mirror}\ t) = t$$

zu beweisen, d.h. wir zeigen, dass

$$R = \{(\mathit{mirror}\ (\mathit{mirror}\ t), t) \mid t \in \mathit{Tree}\}$$

eine Bisimulation ist. Sei also $t \in \mathit{Tree}$; wir rechnen die drei Bedingungen für das entsprechende Paar $\mathit{mirror}\ (\mathit{mirror}\ t)\ R\ t$ wie folgt nach:

- *out*:
$$\mathit{out}\ (\mathit{mirror}\ (\mathit{mirror}\ t)) = \mathit{out}\ (\mathit{mirror}\ t) = \mathit{out}\ t$$
- *left*:
$$\begin{aligned} \mathit{left}\ (\mathit{mirror}\ (\mathit{mirror}\ t)) &= \mathit{mirror}\ (\mathit{right}\ (\mathit{mirror}\ t)) \\ &= \mathit{mirror}\ (\mathit{mirror}\ (\mathit{left}\ t)) \\ &R\,(\mathit{left}\ t). \end{aligned}$$
- *right*: analog.

4.8 Kodatentypen mit Alternativen

Wir betrachten das folgende Beispiel eines Kodatentyps mit Alternativen:

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

Diese Spezifikation definiert die finale G-Koalgebra für den Mengenoperator

$$GX = 1 + A \times X$$

(mit $A = [\![a]\!]$), d.h. $\alpha : N \to 1 + A \times N$.

Hieraus ergibt sich die disjunkte Zerlegung

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

so dass

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

Wir definieren allgemein:

Definition 4.45. Für $GX = \sum_{i=1}^{n} A_i \times X^{k_i}$ beschreiben wir G-Koalgebren durch die *Kosignatur* Σ aus

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

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

  wobei $[\![a_i]\!] = A_i$ und $j = 1, \ldots, k_i$ (im Beispiel: $G = 1 \times X^0 + A \times X^1$, *kein* t_{1j}, $h_1 = \mathit{end}$, $t_{21} = \mathit{tl}$, $h_2 = \mathit{hd}$).

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

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

Definition 4.46. Eine Funktion $f : M \to N$ ist ein Σ-Homomorphismus (schreibe: $f : \mathfrak{M} \to \mathfrak{N}$), wenn sie die folgenden Eigenschaften erfüllt:

- $f[\mathfrak{M}[\![d_i]\!]] \subseteq \mathfrak{N}[\![d_i]\!]$ für alle d_i

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

Man sieht leicht, dass die Σ-Koalgebren im obigen Sinne modulo einfacher Konversionen genau die G-Koalgebren sind. Eine Kodatentypdeklaration mit Kosignatur Σ definiert die finale Σ-Koalgebra; Elemente dieser Koalgebra $[\![\Sigma]\!]$ sind unendliche Bäume, in denen jeder Knoten x sowohl mit einer Alternative d_i als auch mit einem Element $a \in A_i$ beschriftet ist (dann $[\![\Sigma]\!][\![h_i]\!](x) = a$) und über k_i Nachfolgeknoten verfügt (also über einen Nachfolgeknoten y_j pro t_{ij}; dann $[\![\Sigma]\!][\![t_{ij}]\!](x) = y_j$). Für den oben betrachteten Kodatentypen `IList a` besteht die finale Koalgebra also beispielsweise aus allen endlichen oder unendlichen Listen von Elementen aus a. Der Beweis der Finalität läuft letztlich genauso wie der für Streams: Der eindeutige Homomorphismus in die finale Koalgebra bildet einen Zustand x in einer Σ-Koalgebra $\mathfrak{N}$ auf einen unendlichen Baum ab, dessen Beschriftung mit Alternativen (und damit dessen Verzweigung) und mit Elementen der A_i durch wiederholte Anwendung der t_{ij} und (letztlich) der h_i (genauer geschrieben $\mathfrak{N}[\![t_{ij}]\!]$ und $\mathfrak{N}[\![h_i]\!]$) auf x abgelesen werden.

Die Finalität der finalen Σ-Koalgebra $\mathfrak{M}$ läuft dann wie schon im Fall der Streams auf ein korekursives Definitionsprinzip für Funktionen $f : \mathfrak{N} \to \mathfrak{M}$ hinaus. Das Format ist im wesentlichen dasselbe wie bisher; zu beachten ist nur, dass für jedes Element $x \in N$ jeweils anzugeben ist, in welcher Alternative d_i x bzw. äquivalenterweise $f(x)$ liegt; wir deuten dies in der Notation durch $f(x)@d_i$ an.

Man erhält ferner für jede Kosignatur ein Bisimulationsprinzip: R ist eine Bisimulation, wenn aus xRy stets folgt, dass

- x und y zur gleichen Alternative d_i gehören;
- für dieses i dann x und y denselben Output unter h_u liefern; und
- für $j = 1, \ldots, k_i$ die t_{ij}-Nachfolger von x und y wieder in der Relation R stehen.

Wir konkretisieren diese Begriffe in den folgenden Beispielen.

Beispiel 4.47. Wie gehabt definieren wir Funktionen in finale Koalgebren mittels Korekursion. Beispielsweise können wir eine Funktion $\mathit{takeUntil} : a \times \mathit{IList}\, a \to \mathit{IList}\, a$ definieren, so dass $\mathit{takeUntil}\ (x, s)$ eine Liste liefert, die so lange Elemente von s übernimmt, bis das Element x erreicht wird. Falls das Element x erreicht wird, soll $\mathit{takeUntil}\ (x, s)$ sofort enden (ohne x zu übernehmen).

Wir müssen hierzu eine Σ-Koalgebra auf $a \times \mathit{IList}\, a$ angeben, die dann $\mathit{takeUntil}$ als eindeutigen Morphismus in die finale Koalgebra induziert. Hierzu ist zunächst eine disjunkte Zerlegung von $a \times \mathit{IList}\, a$ anzugeben:

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

Im Falle $(\mathit{takeUntil}\ \ x\ s)@\mathit{dead}$ ist dann nichts weiter zu tun (formal ist die Koalgebraoperation h_1 hier gegeben durch $h_1(x, s) = * \in 1$); für $(\mathit{takeUntil}\ \ x\ s)@\mathit{alive}$ definieren wir

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

d.h. unsere Koalgebraoperationen auf der *alive*-Komponente sind

$$h_2\ (x, s) = hd\ s$$
$$t_2\ (x, s) = (x, tl\ s).$$

Beispiel 4.48. In leicht liberalisierter Syntax betrachten wir nun die folgende Spezifikation eines weiteren Kodatentyps mit Alternativen:

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

Wir lassen hier also nun Operationen zu, die auf allen Alternativen definiert sind (hier *objects*), und importieren Datentypdeklarationen. Der zu dieser Kosignatur gehörige Mengenoperator ist

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

wobei $A = [\![a]\!]$ und $A^\star$ die Menge aller endlichen Listen über A bezeichnet. (Das ist nicht wirklich allgemeiner als das bisherige Format, da $A^\star \times (1 + X^2)$ in Bijektion steht mit $A^\star \times X^0 + A^\star \times X^2$, nur syntaktisch bequemer.)

Die finale Koalgebra für G enthält also alle *strikten* Binärbäume (d.h. ein Knoten hat entweder keinen oder zwei Nachfolger), in denen unendlich tiefe Verzweigung zulässig ist während jeder Knoten mit einem Element von `List a` $= A^\star$ beschriftet ist.

Stellen wir uns nun vor, zwei Personen laufen unabhängig voneinander durch zwei Terrains (die gleich sein können), und können miteinander kommunizieren. Sie tauschen sich darüber aus, was sie sehen, und wohin sie als nächstes gehen (links oder rechts). Auf ihrem Weg durch das Terrain könnte es nun passieren, dass sie an einem Punkt verschiedene Dinge (`objects`) sehen; in dem Fall befanden sie sich offensichtlich nicht am „gleichen“ Startpunkt. Oder aber sie führen die Prozedur unendlich lange fort und stellen keine Unterschiede fest. In dem Fall kann man ihre beiden Startpunkte „gleich“ nennen.

Das ist das Prinzip der Bisimulation für diesen Datentyp. Formal ist dies eine Instanz des oben für allgemeine Kosignaturen Σ eingeführten Begriffs:

Definition 4.49. Eine Relation $R \subseteq Terrain\ a \times Terrain\ a$ heisst ($Terrain\ a$-)Bisimulation, wenn für alle $x\ R\ y$ gilt:

1. $objects(x) = objects(y)$,
2. falls $x@stuck$, dann gilt auch $y@stuck$ ($deadend(x) = deadend(y)$ gilt dann automatisch),

3. falls $x@junction$, dann gilt auch $y@junction$ und außerdem:

   (a) $(left\, x)\, R(left\, y)$
   (b) $(right\, x)\, R(right\, y)$.

Der Korrektheitssatz instanziiert sich wie folgt:

Satz 4.50. *Wenn R eine Terrain a-Bisimulation ist und $x\, R\, y$, dann $x = y$.*

Wir können das Prinzip der Bisimulation allgemein etwas liberalisieren, indem wir statt $(left\, x)\, R(left\, y)$ und $(right\, x)\, R(right\, y)$ auch jeweils Gleichheit zulassen. Grundlage dafür ist

Definition 4.51. Eine Relation R ist eine *Bisimulation bis auf Bisimilarität*, wenn sie die Bedingungen aus Definition 4.49 erfüllt, allerdings statt Bedingung 3 die Bedingung

3'. Falls $x@junction$, dann gilt auch $y@junction$ und außerdem:

   (a) $(left\, x)\, R(left\, y)$ oder $left\, x = left\, y$
   (b) $(right\, x)\, R(right\, y)$ oder $right\, x = right\, y$.

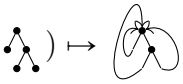
Lemma 4.52. *Wenn R eine Bisimilation bis auf Bisimilarität ist und $x\, R\, y$, dann gilt $x = y$.*

Beweis. Man prüft leicht, dass $R \cup id$ eine Bisimulation ist. □

Beispiel 4.53. Wir möchten nun eine korekursive Funktion

$$loop : (Terrain\, a)@junction \to Terrain\, a$$

definieren, die ein Terrain s auf das Terrain $loop\, s$ abbildet, indem sie alle Sackgassen durch den Ausgangspunkt von s ersetzt, z.B.

$loop :$  $\mapsto$

Nun ist *loop* zunächst nicht in ersichtlicher Weise direkt korekursiv definierbar, da z.B. $left\ (loop\ \ s)$ nicht ohne weiteres wieder von der Form $loop\ \ t$ ist. Wir lösen dieses Problem, indem wir eine Hilfsfunktion *plug* verwenden, so dass $plug\ \ t\ s$ den Graph t so verändert, dass alle „stuck"-Knoten so aussehen und sich so verhalten wie s, wobei wir wiederum $s@junction$ voraussetzen.

Wir definieren *loop* und *plug* wie folgt:

$$loop\, s = plug\ \ s\ \ s$$

$$(plug\ \ t\ \ s)@junction$$

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

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

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

Wir behaupten nun, dass für alle $s \in Terrain\, a@junction$

$$loop\ (loop\ \ s) = loop\ \ s$$

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

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

Sei nun $s \in Terrain\, a@junction$, so dass per Definition $(loop\ (loop\ \ s))\ \ R\ (loop\ \ s)$. Wir überprüfen die drei Bedingungen aus der Definition einer $Terrain\, a$-Bisimulation:

1. Wir haben
$$\begin{aligned} objects\ (loop\ (loop\ \ s)) &= objects\ (plug\ (\underbrace{loop\ \ s}_{@junction})\ (loop\ \ s)) \\ &= objects\ (loop\ \ s) \end{aligned}$$
2. Nach Definition gilt $(loop\ (loop\ \ s))@junction$, d.h. der Fall $(loop\ (loop\ \ s))@stuck$ tritt nicht ein.
3. Nach Definition gilt $(loop\ \ s)@junction$.
   (a) Zu zeigen ist $left\ (loop\ (loop\ \ s))\ \ R\ \ left\ (loop\, s)$. Umformung der linken Seite ergibt
$$\begin{aligned} left\ (loop\ (loop\ \ s)) &= left\ (plug\ (loop\ \ s)\ (loop\ \ s)) \\ &= plug\ (left\ (loop\ \ s))\ (loop\ \ s) \\ &= plug\ (left\ (plug\ \ s\ \ s))\ (loop\ \ s) \\ &= plug\ (plug\ (left\ \ s)\ \ s)\ (loop\ \ s), \end{aligned}$$

und Umformung der rechten Seite

$$\begin{aligned} \mathit{left}(\mathit{loop}\ \ s) &= \mathit{left}(\mathit{plug}\ \ s\ s) \\ &= \mathit{plug}\ (\mathit{left}\ \ s)\ s. \end{aligned}$$

Die beiden Endergebnisse müssten wieder in Relation stehen, was zunächst einmal aber nicht der Fall ist. Wir erweitern daher die ursprüngliche Relation R und setzen

$$R' = \{\ (\mathit{plug}\ (\mathit{plug}\ \ t\ s)\ u,\ \ \mathit{plug}\ \ t\ s)\ \mid \\ t \in \mathit{Terrain}\, a, s, u \in \mathit{Terrain}\, a@\mathit{junction}\} \supseteq R$$

– d.h. wir verallgemeinern unser ursprüngliche Behauptung über *loop* zu einer über *plug*, nämlich, dass *plug* auf Terme der Form $\mathit{plug}\ \ t\ s$ keinen Effekt hat, was intuitiv ebenso klar ist wie unsere ursprüngliche Behauptung, da $\mathit{plug}\ \ t\ s$ keine Zustände der Alternative *stuck* mehr enthalten sollte. Nun gilt insbesondere $\mathit{plug}\ (\mathit{plug}\ (\mathit{left}\ \ s)\ s)\ (\mathit{loop}\ \ s)\ \ R'\ \ \mathit{plug}\ (\mathit{left}\ \ s)\ s$. Wir brechen die Rechnung an dieser Stelle ab und verifizieren stattdessen, dass die neue Relation R' eine Bisimulation ist.

Wir überprüfen dieselben Bedingungen nun also für Paare $(\mathit{plug}\ (\mathit{plug}\ \ t\ s)\ u)\ R'\ (\mathit{plug}\ \ t\ s)$ mit $t \in \mathit{Terrain}\, a$ und $s, u \in \mathit{Terrain}\, a@\mathit{junction}$:

1. Da $(\mathit{plug}\ \ t\ s)@\mathit{junction}$, haben wir $\mathit{objects}\ (\mathit{plug}\ (\mathit{plug}\ \ t\ s)\ u) = \mathit{objects}\ (\mathit{plug}\, t\ s)$.
2. Nach Definition gilt $(\mathit{plug}\ (\mathit{plug}\ t\ s)\ u)@\mathit{junction}$ d.h. der Fall $\mathit{plug}\ (\mathit{plug}\ \ t\ s)\ u@\mathit{stuck}$ tritt nicht ein.
3. (a) Wir haben, wiederum weil $(\mathit{plug}\ \ t\ s)@\mathit{junction}$,

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

und

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

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

(b) Die Rechnung für *right* ist völlig analog.

Beispiel 4.54. Im Folgenden wollen wir einen Kodatentyp *InfTerrain* definieren, der sowohl vom bereits bekannten Typparameter a abhängt, als auch von einem zusätzlichen „Richtungs-Parameter“ d:

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

Dies definiert die finale Koalgebra zu dem Mengenoperator

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

wobei $A = [\![a]\!]$ und $D = [\![d]\!]$; wie schon früher bezeichnet hier $D \to X$ die Menge der Funktionen von D nach X.

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

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

5 Polymorphie und System F

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

```
interface Figure {
        public void draw();
}

class Circle implements Figure {...};
class Triangle implements Figure {...};

public void drawAll (Figure [] figs) {
        for (Figure f : figs)
                f.draw();
}
```

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

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

instance Eq Bool where
    (==) True  True  = True
    (==) False False = True
    (==) _     _     = False
    ...
```

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

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

Die zwei Formen von Polymorphie unterscheiden wir wie folgt:

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

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

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

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

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

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

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

$$2.\quad (\rightarrow_i)\ \frac{\Gamma[x \mapsto \alpha] \vdash\ s : \beta}{\Gamma\ \vdash\ \lambda x.s : \alpha \rightarrow \beta}$$

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

Hier ein Beispiel, in dem ein Ausdruck mehrere Instanzen der polymorphen Funktion $id := \lambda x.x : a \rightarrow a$ verwendet:

```
1 silly :: Bool
2 silly = (λ x y.x) (id True) (id 42)
```

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

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

(wobei wir uns offenhalten, die Typvariable a später geeignet zu instanziieren):

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

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

Definition 5.1 (System-F nach Curry)**.** System F nach Curry (oder λ2-Curry) hat die gleiche Termsprache wie der ungetypte λ-Kalkül (und $\lambda\!\to$-Curry), also $s, t ::= x \mid st \mid \lambda x.s$. *Typen* $\alpha, \beta, \ldots$ in System F werden durch die Grammatik

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

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

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

definiert.

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

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

5.1 Church-Kodierung in System F

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

Natürliche Zahlen:

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

(Man beachte, wie hier die Definition von $\mathbb{N}$ als initialer Datentyp im Typsystem internalisiert wird.))

Paare:

- $(a \times b) := \forall r.\, (a \to b \to r) \to r$
- $pair : \forall ab.\, a \to b \to (a \times b)$
  $pair = \lambda x y f.\, f\ x\ y$
- $fst : \forall ab.\, (a \times b) \to a$
  $fst = \lambda p.\, p\ (\lambda x y.\, x)$
- $snd : \forall ab.\, (a \times b) \to b$
  $snd = \lambda p.\, p\ (\lambda x y.\, y)$

Summen:

- $(a + b) := \forall r.\, (a \to r) \to (b \to r) \to r$
- $inl : \forall ab.\, a \to (a + b)$
  $inl = \lambda \overset{a}{\downarrow} x f g.\, f\ x$
- $inr : \forall ab.\, b \to (a + b)$
  $inr = \lambda \overset{b}{\downarrow} y f g.\, g\ y$
- $case : \forall abs.\, (a \to s) \to (b \to s) \to (a + b) \to s$
  $case = \lambda f g s.\, s\ f\ g$

Summen lassen sich übrigens gut in unserer Syntax für Datentypen darstellen:

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

Listen:

- $List\ a := \forall r.\, r \to (a \to r \to r) \to r$
- $Nil : \forall a.\, List\ a$
  $Nil = \lambda u f.\, u$
- $Cons : \forall a.\, a \to List\ a \to List\ a$
  $Cons = \lambda x l u f.\, f\ x\ (l\ u\ f)$
- $len : \forall a.\, List\ a \to \mathbb{N}$
  $len = \lambda l.\, l\ zero\ (\lambda x r.\, suc\ r)$

Als Beispiel hier die Herleitung der Typisierung von *Suc*:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash n : \forall a.(a \to a) \to a \to a}}{\Gamma \vdash n : (a \to a) \to a \to a}(\forall_e) \quad \dfrac{}{\Gamma \vdash f : a \to a}}{\Gamma \vdash n\ f : a \to a}(\to_e) \quad \dfrac{}{\Gamma \vdash x : a}}{\boxed{\Gamma \vdash n\ f\ x : a}}(\to_e)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : a \to a} \quad \dfrac{\vdots}{\boxed{\Gamma \vdash n\ f\ x : a}}}{\Gamma \vdash f\ (n\ f\ x) : a}(\to_e)}{n : \mathbb{N} \vdash \lambda f\, x\, .\, f\ (n\ f\ x) : (a \to a) \to a \to a}(\to_i) \times 2}{n : \mathbb{N} \vdash \lambda f\, x\, .\, f\ (n\ f\ x) : \mathbb{N}}(\forall_i)}{\vdash \lambda n\, f\, x\, .\, f\ (n\ f\ x) : \mathbb{N} \to \mathbb{N}}(\to_i)$$

mit $\Gamma = \{n : \mathbb{N}, f : a \to a, x : a\}$. Eine durchaus erwartete Eigenschaft ist die Bewahrung von Typisierung durch Reduktion, wie wir sie bereits für $\lambda\to$ gezeigt haben:

Satz 5.3 (Subject Reduction)**.** *Wenn* $\Gamma \vdash s : \alpha$ *und* $s \to_\beta t$*, dann* $\Gamma \vdash t : \alpha$

Wesentlich erstaunlicher ist dagegen folgende Aussage:

Satz 5.4 (Normalisierung, Girard)**.** $\lambda 2$ *ist stark normalisierend; d.h. wenn* $\Gamma \vdash s : \alpha$ *in* $\lambda 2$*, dann ist* s *stark normalisierend.*

Wir verschieben den nichttrivialen Beweis auf Abschnitt 5.4.

Tatsächlich kann jede totale berechenbare Funktion, die in Arithmetik zweiter Stufe definierbar ist, als Term in System F geschrieben werden! Dies schließt ein

- Alle primitiv rekursiven Funktionen
- Die Ackermannfunktion (die bekanntermaßen nicht primitiv rekursiv ist)
- Intuitiv: Compiler sind definierbar, Interpreter nicht (warum letzteres?).

Typinferenz und sogar Typüberprüfung im System F sind *unentscheidbar* (Wells, 1994). Zwei Ansätze zur Lösung sind:

- Einschränkung des Typsystems zur sogenannten ML-Polymorphie (in der dann weniger Terme typisierbar sind)
- Übergang zu einem Church-System.

5.2 Curry vs. Church

Wir erinnern daran, dass sich Church-Systeme des λ-Kalküls von Curry-Systemen dadurch unterscheiden, dass sie eine explizite Typisierung von Variablen in λ-Abstraktionen verlangen; die Church-Variante von $\lambda\!\!\rightarrow$ z.B. hat durch die Grammatik

$$t, s ::= x \mid t\ s \mid \lambda x : \alpha.t$$

definierte Terme, und die Typregel $(\rightarrow_i)$ hat die Form

$$\frac{\Gamma[x \mapsto \alpha] \vdash\ t : \beta}{\Gamma\ \vdash\ \lambda x : \alpha\,.\,t\ \ : \alpha \rightarrow \beta}$$

– d.h. bei Rückwärtsanwendung der Typregeln muss man Typen der Variablen im Kontext nie erfinden, da sie ausdrücklich im Term deklariert sind.

Die Church-Variante von System F (oder $\lambda 2$-*Church*) hat dementsprechend durch die Grammatik

$$t, s ::= x \mid t\ s \mid \lambda x : a.t \mid \Lambda a.t \mid t\ \alpha$$

definierte Terme. Hierbei ist $\Lambda a.\, t$ ein über a explizit polymorpher Term, und Anwendung $t\ \alpha$ eines polymorphen Terms t auf einen Typ α liefert die Instanz von t für den Typ α. Hierfür haben wir eine eigene Form von β-Reduktion:

$$(\Lambda a.t)\alpha \rightarrow_\beta t[\alpha/a].$$

Die Typregeln des Systems sind

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

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

Beispiel 5.5.

1. $\vdash \Lambda a.\, \lambda x : a.\, x : \forall a.\, a \to a$
2. $\vdash \lambda x : \mathbb{N}.\, suc\ (id\ \mathbb{N}\ x) : \mathbb{N} \to \mathbb{N}$
3. $\vdash \Lambda b.\, \lambda x : (\forall a.\, a).\, x\ b : \forall b.\, (\forall a.\, a) \to b$
4. $\vdash \Lambda b.\, \lambda x : (\forall a.\, a).\, x\ ((\forall a.a) \to b)\ x : \forall b.(\forall a.a) \to b$

Die beiden letzten Beispiele demonstrieren das Prinzip *ex falso quodlibet*, d.h. aus einer falschen Annahme kann Beliebiges gefolgert werden. Hierbei spielt der Typ $\forall a.\, a$ die Rolle des *falsum*, das somit als die Aussage „jede Aussage ist wahr“ definiert ist.

5.3 ML-Polymorphie

ML-Polymorphie ist eine Einschränkung von System F á la Curry, die vielen funktionalen Programmiersprachen zu Grunde liegt, u.a. eben ML, aber auch der Grundversion von Haskell (die Glasgow Extensions verwenden volles System F und mehr). In ML-Polymorphie ist Typinferenz entscheidbar, durch einen ganz ähnlichen Algorithmus, wie wir ihn schon für $\lambda\!\to$ verwendet haben. Im einzelnen haben wir folgende Anpassungen:

- $\forall$ ist nur auf oberster Ebene von Typausdrücken erlaubt (d.h. es darf nicht im linken Argument von $\to$ auftauchen).

  Beispielsweise ist der Typ $(\forall a.\, a) \to b$ aus obigem Beispiel nicht mehr erlaubt, sehr wohl dagegen der Typ der Identität, $\forall a.\, a \to a$.
- Mehrfachinstanziierung polymorpher Funktionen ist nur per *let* erlaubt (daher wird ML-Polymorphie auch als *let-Polymorphie* bezeichnet):

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

Formal definieren wir den *ML-polymorphen λ-Kalkül* wie folgt. Wir unterscheiden zwischen *Typen*, definiert durch die Grammatik

$$\alpha, \beta ::= a \mid \alpha \to \beta,$$

und *Typschemata*, gegeben durch die nicht-rekursive Grammatik

$$S ::= \forall a_1, \ldots, a_k.\alpha \qquad (k \geqslant 0).$$

(Insbesondere sind Typen Typschemata, aber nicht umgekehrt.) Für Terme führen wir zusätzlich das let-Konstrukt ein, d.h. Terme sind gegeben durch die Grammatik

$$t, s ::= x \mid t\, s \mid \lambda x.t \mid \text{let } x = t \text{ in } s.$$

Kontexte haben die Form $\Gamma = (x_1 : S_1, \ldots, x_n : S_n)$ mit paarweise verschiedenen Variablen x_i, d.h. weisen Variablen Typschemata zu. Typisierungsurteile sind weiterhin von der Form $\Gamma \vdash t : \alpha$, wobei wir Typvariablen, die im Typ α, aber nicht im Kontext Γ vorkommen, als implizit allquantifiziert lesen, d.h. für $FV(\alpha)\backslash FV(\Gamma) = \{a_1, \ldots, a_k\}$ setzen wir

$$Cl(\Gamma, \alpha) = \forall a_1. \ldots \forall a_k. \alpha.$$

Z.B. haben wir wie bisher $\vdash \lambda x.x : a \to a$, und nach obiger Definition gilt $Cl\big((), a \to a\big) = \forall a.a \to a$.

Die Typisierungsregeln sind wie folgt:

- $(\forall_e)\ \dfrac{}{\Gamma \vdash x : \alpha[\beta_1/a_1, \ldots, \beta_k/a_k]} \quad (x : \forall a_1. \ldots \forall a_k. \alpha) \in \Gamma$

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

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

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

(Man beachte insbesondere, dass die Regel $(\forall_e)$ effektiv die gleichnamige Regel aus System F mit der (Ax)-Regel kombiniert; dies ist notwendig, da Termen keine Typschemata zugewiesen werden können.)

Beispiel 5.6. Unseren Beispielterm von oben können wir in diesem System typisieren. Eine naive Suche nach einer Typherleitung verläuft etwa wie folgt:

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

Wir können dann z.B. $\gamma = b \to b$, $\beta = b \to b$ wählen.

Achtung: Intuitiv denkt man, dass man $\text{let}\, id = \lambda x.\, x \text{ in } id\ id$ kürzer auch als $(\lambda id.\, id\ id)\ \lambda x.\, x$ schreiben kann. Der geklammerte Ausdruck ist aber nicht typisierbar, da durch λ gebundene Variablen einen *Typ* haben müssen – nur let lässt auch die Bindung polymorpher, also mit eine Typschema typisierter Variablen zu.

Typinferenz unter ML-Polymorphie ist wie versprochen durch eine Erweiterung unseres Algorithmus für $\lambda\to$ entscheidbar:

Wir haben wieder ein Inversionslemma, das für λ-Abstraktion und Applikation wie für $\lambda\to$ lautet; für Variablen haben wir leicht verallgemeinert

- Wenn $\Gamma \vdash x : \alpha$, dann existieren Typen β_i, und ein Typschema $S = \forall a_1 \ldots \forall a_k . \gamma$, so dass $(x : S) \in \Gamma$ und $\alpha = \gamma[\beta_1/a_1, \ldots, \beta_k/a_k]$.

Für das neue let-Konstrukt schließlich haben wir

- Wenn $\Gamma \vdash (\text{let } x = s \text{ in } t) : \alpha$, dann existiert ein Typ β mit $\Gamma \vdash s : \beta$ und $\Gamma, x : Cl(\Gamma, \beta) \vdash t : \alpha$.

(Der Beweis ist jeweils ebenso leicht wie im Fall von $\lambda\!\rightarrow$.) Hieraus ergibt sich unmittelbar folgende Anpassung von Algorithmus W von Hindley und Milner (von Damas und Milner (1982) wurde der Algorithmus für den ML-polymorphen λ-Kalkül formuliert). Wir erinnern daran, dass $PT(\Gamma; t; \alpha)$ ein Gleichungssystem auf Typen liefert, dessen allgemeinste Lösung σ gleichzeitig die allgemeinste Lösung von $\Gamma\sigma \vdash t : \alpha\sigma$ ist. Wir behalten in der rekursiven Definition die Klauseln für Applikation und λ-Abstraktion bei. Die Klausel für Variablen verallgemeinern wir für $(x : \forall a_1 \ldots \forall a_k . \gamma) \in \Gamma$ zu

$$PT(\Gamma; x; \alpha) = \{\alpha \doteq \gamma[a'_1/a_1, \ldots, a'_k/a_k]\}$$

für *frische* Variablen a'_i; damit stellen wir sicher, dass wir die a_i (jetzt a'_i) bei verschiedenen Vorkommen von x verschieden instanziieren können. Für das neue let-Konstrukt haben wir das Problem, dass die Typvariable, die wir als Typ für s in $\text{let } x = s \text{ in } t$ einführen, keine Berechnung von $Cl(\Gamma, \cdot)$ erlauben würde. Wir müssen das Teilproblem für s also zunächst tatsächlich lösen, d.h. die zugehörige Gleichungsmenge unifizieren. Damit ergibt sich

$$PT(\Gamma; (\text{let } x = s \text{ in } t); \alpha) = PT(\Gamma\sigma[x \mapsto Cl(\Gamma\sigma, \sigma(b))]; t; \alpha\sigma),$$

wobei

$$\sigma = mgu(PT(\Gamma; s; b))$$

mit b frisch.

Beispiel 5.7. Wir führen die Typinferenz für den oben bereits typisierten Term $\text{let } id = \lambda x. x \text{ in } id\ id$ durch (im leeren Kontext ()): Wir berechnen wie in $\lambda\!\rightarrow$

$$mgu(PT((); \lambda x. x; b) = [a \rightarrow a/b].$$

Wir berechnen dann gemäß der Klausel für let mit $\Gamma = (id : \forall a. a \rightarrow a)$

$$\begin{aligned} PT(\Gamma; id\ id; d) &= PT(\Gamma; id; c \rightarrow d) \cup PT(\Gamma; id; c) \\ &= \{c \rightarrow d \doteq a' \rightarrow a'\} \cup \{c \doteq a'' \rightarrow a''\}, \end{aligned}$$

mit mgu $[a'' \rightarrow a''/c, a'' \rightarrow a''/d]$, d.h. der Prinzipaltyp von $id\ id$ ist $a'' \rightarrow a''$.

Der Vollständigkeit halber führen wir noch den induktiven Beweis der Behauptung, dass eine Typsubstitution σ genau dann $\Gamma \vdash t : \alpha$ löst, wenn sie zu einem Unifikator von $PT(\Gamma; t; \alpha)$ erweiterbar ist (Beweis von Satz 3.45), nach. Die Implikation von rechts nach

links ist (wieder) im wesentlichen trivial; wir zeigen die Implikation von links nach rechts, wie gehabt per Induktion über t:

Der Fall, dass t eine Variable x ist, läuft jetzt wie folgt. Nach dem Inversionslemma folgt aus $\Gamma\sigma \vdash x : \alpha\sigma$, dass Typen β_i und ein Typschema $S = \forall a_1 \ldots \forall a_k.\, \gamma$ existieren, so dass $(x : S) \in \Gamma$ und $\alpha\sigma = \gamma[\beta_1/a_1, \ldots, \beta_k/a_k]$. Dann erweitern wir σ zu einer Lösung von $PT(\Gamma; x; \alpha) = \{\alpha \doteq \gamma[a'_1/a_1, \ldots, a'_k/a_k]$, indem wir jeweils a'_i durch β_i substituieren.

Es bleibt der Fall für das neue let-Konstrukt; hierfür benötigen wir noch etwas Maschinerie. Wir definieren zunächst eine Allgmeinheitsordnung auf Typschemata:

Definition 5.8. Für ein Typschema $S = \forall a_1, \ldots, a_n(\alpha)$ is die Menge $\mathsf{Inst}(S)$ der *Instanzen* von S definiert als

$$\mathsf{Inst}(S) = \{\alpha[\beta_1/a_1, \ldots, \beta_n/a_n] \mid \beta_1, \ldots, \beta_n \text{ Typen}\}.$$

Damit ist S *allgemeiner* als ein Typschema S', wenn $\mathsf{Inst}S' \subseteq \mathsf{Inst}S$.

Lemma 5.9. *Für Typschemata $S = \forall a_1, \ldots, a_n(\alpha)$ und $S' = \forall b_1, \ldots, b_k(\beta)$ sind äquivalent:*

1. *S ist allgemeiner als S';*
2. *$\beta \in \mathsf{Inst}(S)$ und $b_i \notin FV(S)$ für $i = 1, \ldots, k$.*

Beweis. 1. $\Longrightarrow$ 2.: Wir haben $\beta \in \mathsf{Inst}(S')$ und damit per 1. auch $\beta \in \mathsf{Inst}(S)$. Für den zweiten Teil der Behauptung betrachte man $\beta[b'_1/b_1, \ldots, b'_k/b_k] \in \mathsf{Inst}(S')$ und beachte, dass alle Instanzen von S alle freien Variablen von S enthalten.

2. $\Longrightarrow$ 1.: Nach Voraussetzung hat β die Form $\beta = \alpha[\gamma_1/a_1, \ldots, \gamma_n/a_n]$ für Typen $\gamma_1, \ldots, \gamma_n$. Seien nun $\delta_1, \ldots, \delta_k$ Typen und $\sigma = [\delta_1/b_1, \ldots, \delta_k/b_k]$. Wegen $b_i \notin FV(S)$ gilt für die hierdurch bestimmte Instanz $\beta\sigma$ von S' dann

$$\beta\sigma = \alpha[\gamma_1\sigma/a_1, \ldots, \gamma_n\sigma/a_n],$$

d.h. $\beta\sigma \in \mathsf{Inst}(S)$. □

Lemma 5.10. *$Cl(\Gamma, \alpha)\sigma$ ist allgemeiner als $Cl(\Gamma\sigma, \alpha\sigma)$.*

Beweis. Wir verwenden die obige Charakterisierung.

- Es ist klar, dass $\alpha\sigma \in \mathsf{Inst}(Cl(\Gamma, \alpha)\sigma)$ – der unter den Quantoren stehende Teil von $Cl(\Gamma, \alpha)\sigma$ unterscheidet sich von $\alpha\sigma$ nur dadurch, dass die Wirkung von σ auf $Cl(\Gamma, \alpha)$ teilweise durch in $Cl(\Gamma, \alpha)$ gebundene Variablen blockiert sein kann; durch Nachholen der blockierten Ersetzungen wird $\alpha\sigma$ eine Instanz von $Cl(\Gamma, \alpha)\sigma$.
- Sei nun b eine freie Variable in $Cl(\Gamma, \alpha)\sigma$, d.h. $b \in FV(\sigma(a))$ für ein $a \in FV(Cl(\Gamma, \alpha)) \subseteq FV(\Gamma)$. Es folgt $b \in FV(\Gamma\sigma)$, d.h. b wird in $Cl(\Gamma\sigma, \alpha\sigma)$ nicht gebundenen. □

Nach der Definition der Allgemeinheitsrelation auf Typschemata zeigt man nun sehr leicht

Lemma 5.11 (Abschwächung)**.** *Wenn S allgemeiner als S' ist und $\Gamma, x : S' \vdash t : \alpha$, dann auch $\Gamma, x : S \vdash t : \alpha$.*

Wir setzen damit die Ergänzung des Induktionsbeweises fort, indem wir den fehlenden Fall für das let-Konstrukt durchführen. Sei also $\Gamma\sigma \vdash \text{let } x = s \text{ in } t : \alpha\sigma$. Nach dem Inversionslemma existiert dann ein Typ β mit $\Gamma\sigma \vdash s : \beta$ und

$$\Gamma\sigma, x : Cl(\Gamma\sigma, \beta) \vdash t : \alpha\sigma. \tag{16}$$

Nach Induktionsvoraussetzung ist für frisches b die Substitution $\sigma[b \mapsto \beta]$ erweiterbar zu einem Unifikator σ' von $PT(\Gamma; s; b)$. Sei nun $\tau = mgu(PT(\Gamma\sigma; s; b))$. Dann existiert $\bar{\sigma}$ mit $\tau\bar{\sigma} = \sigma'$; der Unifikationsalgorithmus aus GLoIn konstruiert τ sogar so, dass $\tau\sigma' = \sigma'$. Wir müssen zeigen, dass σ' erweiterbar zu einem Unifikator von $PT(\Gamma\tau, x : Cl(\Gamma\tau, \tau(b)); t; \alpha\tau)$ ist; dazu reicht nach Induktionsvoraussetzung zu zeigen, dass $\Gamma\tau\sigma', x : Cl(\Gamma\tau, \tau(b))\sigma' \vdash t : \alpha\tau\sigma'$, d.h. nach der erwähnten Zusatzeigenschaft von τ sowie der Tatsache, dass σ' eine Erweiterung von σ ist,

$$\Gamma\sigma, x : Cl(\Gamma\tau, \tau(b))\sigma' \vdash t : \alpha\sigma.$$

Dies folgt per Abschwächung aus (16): Nach Lemma 5.10 ist $Cl(\Gamma\tau, \tau(b))\sigma'$ allgemeiner als $Cl(\Gamma\tau\sigma', \tau(b)\sigma') = Cl(\Gamma\sigma', \sigma'(b)) = Cl(\Gamma\sigma, \beta)$, wobei wir erneut $\tau\sigma' = \sigma'$ verwenden.

5.4 Starke Normalisierung in System F

Wir holen nunmehr den aufgeschobenen Beweis des Normalisierungssatzes für System F ($\lambda 2$) nach, d.h. wir zeigen, dass jeder im System typisierbare Term stark normalisierend ist. Der Beweis verallgemeinert den, den wir weiter oben für $\lambda\!\!\rightarrow$ angegeben haben; wir geben ihn dennoch hier vollständig an, mit wörtlicher Wiederholung der gemeinsamen Teile.

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

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

wobei Λ die Menge aller λ-Terme ist und ξ eine *Typbelegung*

$$\xi : \mathbf{V} \to SAT,$$

die also jede Typvariable durch eine *saturierte* Menge von stark normalisierenden Termen im Sinne von Definition 5.12 unten interpretiert. Wenn wir Korrektheit des Typsystems bezüglich dieser Semantik zeigen, d.h wenn wir zeigen, dass jeder typisierbare Term tatsächlich zur Interpretation seines Typs gehört, ist das Resultat offenbar bewiesen. Wir geben die Semantik rekursiv an: Für $A, B \subseteq \Lambda$ schreiben wir

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

und setzen dann

$$\begin{aligned} [\![a]\!]_\xi &= \xi(a) \\ [\![\alpha \to \beta]\!]_\xi &= [\![\alpha]\!]_\xi \to [\![\beta]\!]_\xi \\ [\![\forall a.\, \alpha]\!]_\xi &= \bigcap_{A \in SAT} [\![\alpha]\!]_{\xi[a \mapsto A]}. \end{aligned}$$

Die noch ausstehende Definition von Saturiertheit lautet wie folgt.

Definition 5.12. Eine Teilmenge $A \subseteq SN$ heißt *saturiert*, wenn

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

Wir setzen dann

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

Lemma 5.13 (Saturiertheitslemma)**.**

a) $SN \in SAT$.

b) $[\![\alpha]\!]_\xi \in SAT$ *für alle* α, ξ.

Beweis. a):

1. Seien x eine Variable und $t_1, \dots, t_n \in SN$. Zu zeigen ist $xt_1 \dots t_n \in SN$. Das ist aber klar: Jeder Redukt von $xt_1 \dots t_n$ ist von der Form $xt_1' \dots t_n'$ mit $t_i \to_\beta^\star t_i'$ (also $t_i' \in SN$), so dass man aus einer unendlichen Reduktionssequenz für $xt_1 \dots t_n$ auch eine für eines der t_i gewinnen würde, im Widerspruch zu $t_i \in SN$.
2. Seien $s \in SN$ und $t[s/x]u_1 \dots u_n \in SN$; dann gilt $t, u_1, \dots, u_n \in SN$. Zu zeigen ist $v := (\lambda x.\, t)su_1 \dots u_n \in SN$. Da $t, s, u_1, \dots, u_n \in SN$, hat jede unendliche Reduktionssequenz von v die Form

   $$(\lambda x.\, t)su_1 \dots u_n \to_\beta^\star (\lambda x.\, t')s'u_1' \dots u_n' \to_\beta t'[s'/x]u_1' \dots u_n' \to_\beta \dots$$

   (d.h. muss irgendwann den Redex $(\lambda x.\, t)s$ reduzieren). Da aber $t[s/x]u_1 \dots u_n \to_\beta^\star t'[s'/x]u_1' \dots u_n'$ (eventuell per Reduktion von mehreren Vorkommen von s in $t[s/x]$ ähnlich wie im Beweis des Critical Pair Lemma), ist dies ein Widerspruch zu $t[s/x]u_1 \dots u_n \in SN$.

b): Induktion über α.

1. Für eine Typvariable a gilt $[\![a]\!]_\xi = \xi(a) \in SAT$.
2. Seien $A := [\![\alpha]\!]_\xi, B := [\![\beta]\!]_\xi$ saturiert; zu zeigen ist, dass $A \to B$ saturiert ist.
   (a) $A \to B \subseteq SN$: Sei $t \in A \to B$. Sei x eine Variable. Da A saturiert ist, gilt $x \in A$, somit $tx \in B$ per Definition von $A \to B$, also $tx \in SN$ und somit $t \in SN$.
   (b) Seien $r_1, \dots, r_n \in SN$; zu zeigen ist $xr_1 \dots r_n \in A \to B$. Sei also $s \in A \subseteq SN$, zu zeigen ist dann $xr_1 \dots r_n s \in B$. Da $s \in SN$, folgt dies aus Saturiertheit von B.
   (c) Seien $s \in SN$ und $t[s/x]r_1 \dots r_n \in A \to B$. Zu zeigen ist $(\lambda x.\, t)sr_1 \dots r_n \in A \to B$. Sei also $v \in A$, zu zeigen ist dann $(\lambda x.\, t)sr_1 \dots r_n v \in B$. Per Saturiertheit von B reicht dazu $t[s/x]sr_1 \dots r_n v \in B$, was aus $v \in A$ und $t[s/x]r_1 \dots r_n \in A \to B$ folgt.

3. Für den Fall $\forall a.\, \alpha$ reicht es anzumerken, dass SAT offensichtlich abgeschlossen unter großen Durchschnitten *nichtleerer* Mengensysteme ist (der Durchschnitt des leeren Mengensystems wäre die Menge aller Terme, die keine Teilmenge von SN ist); aus Behauptung a) des Lemmas folgt, dass im Durchschnitt $\bigcap_{A \in SAT}$ mindestens ein Index A, nämlich $A = SN$, vorkommt. □

Lemma 5.14 (Substitutionslemma)**.** *Wir haben* $[\![\alpha[\beta/a]]\!]_\xi = [\![\alpha]\!]_{\xi[a \mapsto [\![\beta]\!]_\xi]}$.

Beweis. Induktion über α. □

Definition 5.15 (Erfülltheit/Konsequenz)**.** Sei σ eine Substitution und ξ eine Typbelegung. Dann schreiben wir

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

Lemma 5.16 (Korrektheit)**.** *Wenn* $\Gamma \vdash t : \alpha$*, dann* $\Gamma \models t : \alpha$.

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

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

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

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

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

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

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

Sei $\Gamma[x \mapsto \alpha] \models t : \beta$ und $\sigma, \xi \models \Gamma$. Zu zeigen ist $(\lambda x.\, t)\sigma \in [\![\alpha]\!]_\xi \to [\![\beta]\!]_\xi$. Sei also $v \in [\![\alpha]\!]_\xi$; zu zeigen ist dann $((\lambda x.\, t)\sigma)v \in [\![\beta]\!]_\xi$. Wir können ohne Einschränkung annehmen, dass $\sigma(x) = x$ und das x frisch für σ ist, so dass $(\lambda x.\, t)\sigma = \lambda x.\, t\sigma$. Damit rechnen wir wie folgt:

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

letzteres gilt nach Voraussetzung.

$$(\forall_e)\ \frac{\Gamma \vdash t : \forall a.\, \alpha}{\Gamma \vdash t : \alpha[\beta/a]}$$

Korrektheit dieser Regel ist klar, da

$$[\![\forall a.\, \alpha]\!]_\xi = \bigcap_{A \in SAT} [\![\alpha]\!]_{\xi[a \mapsto A]} \subseteq [\![\alpha]\!]_{\xi[a \mapsto [\![\beta]\!]_\xi]} = [\![\alpha[\beta/a]]\!]_\xi,$$

wobei wir das Saturiertheitslemma für β und im letzten Schritt das Substitutionslemma verwenden.

$$(\forall_i)\ \frac{\Gamma \vdash t : \alpha}{\Gamma \vdash t : \forall a.\, \alpha}\ (a \notin FV(\Gamma))$$

Sei $\Gamma \models t : \alpha$ und $\sigma, \xi \models \Gamma$. Da $a \notin FV(\Gamma)$ und somit $\xi(a)$ für die Erfülltheit von Γ keine Rolle spielt, gilt dann auch $\sigma, \xi[a \mapsto A] \models \Gamma$ für alle $A \in SAT$. Es folgt gemäß der Prämisse der Regel $\sigma, \xi[a \mapsto A] \models t : \alpha$ und somit $t\sigma \in [\![\alpha]\!]_{\xi[a \mapsto A]}$ für alle $A \in SAT$, d.h. $t\sigma \in [\![\forall a.\, \alpha]\!]_\xi$, also $\sigma, \xi \models t : \forall a.\, \alpha$. □

Daraus folgt im wesentlichen sofort unser Zielresultat:

Satz 5.17. *$\lambda 2$ ist stark normalisierend.*

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

6 Reguläre Ausdrücke und endliche Automaten

6.1 Recall: Nichtdeterministische endliche Automaten

(Siehe BFS.)

Definition 6.1. Ein *nichtdeterministischer endlicher Automat mit ϵ-Transitionen* (NFA^ϵ) ist ein Tupel

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

bestehend aus

- einer endlichen Menge Q von *Zuständen*;
- einem endlichen *Alphabet* Σ;
- einer *Transitionsabbildung* $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ (man erinnere sich, dass $\mathcal{P}(X)$ die *Potenzmenge* einer Menge X bezeichnet, also die Menge aller Teilmengen von X);
- einem *Anfangszustand* $s \in Q$;
- einer Menge $F \subseteq Q$ von *akzeptierenden* oder *finalen* Zuständen.

Wir schreiben

$$q \xrightarrow{a} q : \iff q' \in \Delta(q, a).$$

Definition 6.2. Definiere $q \overset{u}{\Longrightarrow} q'$ für Wörter $u \in \Sigma^\star$ induktiv per

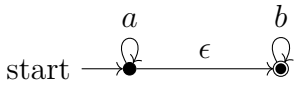
$$\frac{}{q \overset{\epsilon}{\Longrightarrow} q}$$

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

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

Wir sagen dann, dass A ein Wort u *akzeptiert*, wenn ein akzeptierender Zustand $q \in F$ existiert mit $s \overset{u}{\Longrightarrow} q$. Eine *Sprache* ist naheliegenderweise eine Menge von Worten (über Σ). Die von A *akzeptierte Sprache* $L(A)$ ist dann die Menge aller von A akzeptierten Wörter. Eine Sprache $L \subseteq \Sigma^\star$ heißt *regulär*, wenn ein NFA^ϵ A mit $L = L(A)$ existiert, d.h. wenn A L akzeptiert.

Beispiel 6.3. In Abbildungen von Automaten bezeichnen wir den initialen Zustand mit einem eingehenden Pfeil und finale Zustände durch einen zusätzlichen Kringel. Sei etwa A der NFA^ϵ



Dann haben wir

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

Definition 6.4. Ein NFA$^\epsilon$ A ist ein *nichtdeterministischer endlicher Automat (NFA)* genau dann, wenn A keine ϵ-Transitionen hat, d.h. $\Delta(q, \epsilon) = \{\}$ für alle Zustände q.

Ein NFA A ist *deterministisch* (DFA), wenn $|\Delta(q, a)| = 1$ für alle Zustände q. Wir schreiben dann $\Delta(q, a) =: \{\delta(q, a)\}$

Hinsichtlich Ausdrucksstärke sind DFA, NFA und NFA$^\epsilon$ äquivalent:

Lemma 6.5 (Potenzmengenkonstruktion)**.** *Zu jedem NFA$^\epsilon$ A existiert ein DFA B mit $L(B) = L(A)$*

6.2 Reguläre Ausdrücke

Definition 6.6. *Reguläre Ausdrücke $r, s, \ldots$* sind durch die Grammatik

$$r, s ::= 1 \mid 0 \mid r + s \mid rs \mid r^\star \mid a \qquad (a \in \Sigma)$$

definiert. Die Semantik eines regulären Ausdrucks r ist die wie folgt rekursiv definierte Sprache $L(r)$:

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

Beispiel 6.7. Wir können die aus allen Wörtern über $\Sigma = \{a, b\}$, die höchstens zwei aufeinanderfolgende a enthalten, durch den regulären Ausdruck

$$(b + ab + aab)^\star(1 + a + aa)$$

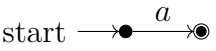
(in `grep`-Notation: `(b|ab|aab)*(|a|aa)`) beschreiben.

Satz 6.8 (Satz von Kleene)**.** *Eine Sprache L ist genau dann regulär, wenn es einen regulären Ausdruck r gibt mit $L = L(r)$*

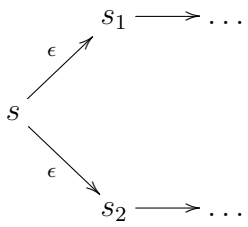
Beweis. „⇐“: Per Induktion über r:

- $r = 0$:
  start ⟶●
- $r = 1$:
  start ⟶◉

- $r = a$:

- $r = r_1 + r_2$: Nach IV haben wir Automaten A_i mit Anfangszuständen s_i, die $L(r_i)$ akzeptieren. Damit akzeptiert der Automat, der als disjunkte Vereinigung der A_i und hinzufügen eines neuen Anfangszustands s mit

  entsteht, offenbar $L(r_1) + L(r_2) = L(r_1 + r_2)$.
- $r = r_1 r_2$: Nach IV haben wir Automaten A_i mit Anfangszuständen s_i, die $L(r_i)$ akzeptieren. Wir konstruieren einen Automaten A ausgehend von der disjunkten Vereinigung von A_1 und A_2. Der initiale Zustand von A ist der initiale Zustand s_1 von A_1, und die akzeptierenden Zustände von A sind die von A_2. Für fügen ferner von jedem akzeptierenden Zustand von A_1 einenq ϵ-Übergang in den initialen Zustand von A_2 hinzu. Dann akzeptiert A gerade $L(r_1 r_2) = L(r_1)L(r_2)$: Um von A akzeptiert zu werden, muss ein Wort, ausgehend von s_1, einen finalen Zustand von A_2 erreichen, was nach Konstruktion gerade über einen finalen Zustand von A_1 und den initialen Zustand von A_2 möglich ist, so dass r aus einem von A_1 akzeptierten Wort gefolgt von einem von A_2 akzeptierten Wort besteht.
- für $r = r_0^\star$: Nach IV haben wir einen NFA$^\epsilon$ A_0 mit $L(A) = r_0$. Wir erweitern A_0 zu einem NFA$^\epsilon$ A, indem wir einen neuen Zustand s als initialen Zustand hinzufügen, den wir gleichzeitig als einzigen finalen Zustand erklären, und ϵ-Transitionen von s in den initialen Zustand von A_0 sowie von den finalen Zuständen von A_0 nach s. Dann akzeptiert A gerade $L(r^\star)$: Um von A akzeptiert zu werden, muss ein Wort u entweder leer sein oder akzeptiert werden, nachdem es einmal A_0 passiert hat und über einen finalen Zustand von A_0 wieder nach s gekommen ist, d.h. u muss akzeptiert werden, nachdem man ein nichtleeres Präfix aus $L(r)$ entfernt hat; per Induktion über die Länge von u folgt dann, dass der Rest von u in $L(r^\star)$ liegt, also auch u.

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

($*$): Für alle $q, q' \in Q$ und alle $R \subseteq Q$ gibt es einen regulären Ausdruck $r^R_{q,q'}$ mit

$$L(r^R_{q,q'}) = \{u \in \Sigma^\star \mid q \xrightarrow{u} q' \text{ mit Zwischenzuständen nur aus } R\}.$$

Daraus folgt dann die Behauptung, denn $L(A) = \sum_{q \in F} r^Q_{s,q}$.

Beweis von ($*$) per Induktion über $|R|$:

- Induktionsanfang ($|R| = 0$):
  - Fall 1: $q \neq q'$: $r^{\emptyset}_{q,q'} = \sum_{q \xrightarrow{a} q'} a$
  - Fall 2: $q = q'$: $r^{\emptyset}_{q,q'} = 1 + \sum_{q \xrightarrow{a} q'} a$
- Induktionsschritt: Sei $|R| > 0$. Wähle dann $q_0 \in R$, $R_0 := R \backslash \{q_0\}$. Wir haben nach Induktionsvoraussetzung schon alle $r^{R_0}_{q,q'}$ konstruiert. Wir setzen dann

$$r^{R}_{q,q'} := r^{R_0}_{q,q'} + r^{R_0}_{q,q_0} (r^{R_0}_{q_0,q_0})^{\star} r^{R_0}_{q_0,q'}.$$

  Um zu sehen, dass $r^{R}_{q,q'}$ bei dieser Definition die in $(*)$ angegebene Sprache akzeptiert, gehen wir per Fallunterscheidung vor. Ein Weg per u von q nach q' mit Zwischenzuständen nur in R kann entweder q_0 passieren oder nicht. Letzterer Fall wird per Induktion gerade durch den linken Summanden $r^{R_0}_{q,q'}$ beschrieben, ersterer durch den rechten Summanden $r^{R_0}_{q,q_0}(r^{R_0}_{q_0,q_0})^{\star} r^{R_0}_{q_0,q'}$: wir unterteilen den Weg jedes Mal, wenn er q_0 passiert, und konkatenieren die so entstehenden Teilwege, die jetzt nur noch durch Zwischenzustände in R_0 laufen.

□

Bemerkung 6.9. Aus dem Satz von Kleene folgt sofort, dass reguläre Ausdrücke unter Durchschnitt und Komplement von Sprachen abgeschlossen sind, da wir diese Konstruktionen auf DFA leicht realisieren können (wir komplementieren einen DFA, indem wir einfach die akzeptierenden Zustände zu nicht akzeptierenden machen und umgekehrt; Durchschnitte können wir in der bekannten Weise mittels Komplement und Vereinigung konstruieren: $r \cap s = \neg(\neg r + \neg s)$).

Wir merken an, dass Komplementierung von NFA nicht so einfach ist wie für DFA – wenn man z.B. die obige Konstruktion zur Komplementierung von DFAs auf einen NFA A für r anwendet, akzeptiert der so entstehende NFA B nicht unbedingt das Komplement von r: ein Wort $u \in L(r)$ kann in A durchaus auch nicht-finale Zustände erreichen (damit u von A akzeptiert wird, reicht es, dass u irgendeinen finalen Zustand erreicht, u kann aber auch andere Zustände erreichen), und würde dann von B akzeptiert. Um einen NFA zu komplementieren, wird man ihn daher typischerweise zunächst per Potenzmengenkonstruktion in einen DFA transformieren.

Durchschnittsbildung dagegen funktioniert ohne weiteres auch für NFA: Gegeben seien nichtdeterministische Automaten $A = (Q_A, \Sigma, \Delta_A, s_A, F_A)$ und $B = (Q_B, \Sigma, \Delta_B, s_B, F_B)$. Wir definieren dann einen *Produktautomaten*

$$A \times B := (Q_A \times Q_B, \Sigma, \Delta, (s_A, s_B), F_A \times F_B)$$

mit

$$\Delta(a, (p, q)) = \underbrace{\Delta_A(a, p)}_{\subseteq Q_A} \times \underbrace{\Delta(a, q)}_{\subseteq Q_B} \subseteq Q_A \times Q_B.$$

D.h. wir definieren finale Zustände und Transitionen in $A \times B$ durch

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

Man sieht leicht, dass $L(A \times B) = L(A) \cap L(B)$.

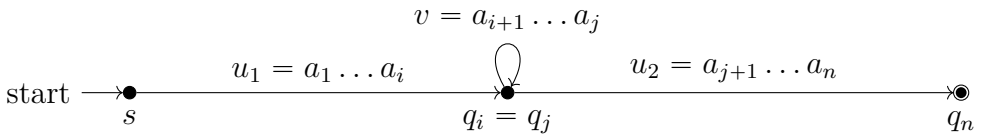
Satz 6.10 (Pumping-Lemma)**.** *Wenn eine Sprache L regulär ist, dann existiert $l \geqslant 1$, so dass für alle $w \in L$ mit $|w| \geqslant l$ eine Unterteilung $w = u_1 v u_2$ mit $|v| \geqslant 1$, $|u_1 v| \leqslant l$ existiert, so dass*

$$u_1 v^\star u_2 \subseteq L.$$

Wir verwenden den Satz vor allem in kontraponierter Form:

> *Wenn für alle $l \geqslant 1$ ein $w \in L$ mit $|w| \geqslant l$ existiert, so dass*
>
> $$u_1 v^\star u_2 \not\subseteq L$$
>
> *für alle Unterteilungen $w = u_1 v u_2$ mit $|v| \geqslant 1$ und $|u_1 v| \leqslant l$, dann ist L* nicht *regulär.*

Beweis (Satz 6.10). Sei $A = (Q, \Sigma, \delta, s, F)$ ein DFA mit $L(A) = L$. Setze dann $l = |Q|$. Sei $w \in L$ mit $|w| \geqslant l$, also $w = a_1 \dots a_n$ mit $n \geqslant l$. Dann $w \in L(A)$, also existiert $s = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ mit q_n final. Das sind mindestens $l + 1$ Zustände, also existieren $i < j \leqslant l$ mit $q_i = q_j$. Wir unterteilen u gerade bei i und j:



Nach Erreichen von q_i mittels u_1 können wir die Schleife bei $q_i = q_j$ mittels v beliebig oft durchlaufen (auch kein Mal) und erreichen dann letztlich mittels u_2 den finalen Zustand q_n, d.h. $u_1 v^\star u_2 \subseteq L$ wie verlangt. □

Beispiel 6.11. 1. $L = \{a^n b^n \mid n \geqslant 0\}$ ist nicht regulär, denn:

> Sei $l \geqslant 1$ gegeben; wähle dann $w = a^l b^l$. Sei eine Zerlegung $w = u_1 v u_2$ mit $|v| \geqslant 1$, $|u_1 v| \leqslant l$ gegeben. Dann haben wir $v \in a^+$, und damit $u_1 v^k u_2 = u_1 u_2 \notin L$ für $k \neq 1$, da Hinzufügen von Kopien von v oder Weglassen von v die Anzahl as ändert, Die Anzahl bs aber nicht.

2. $L = \{u \in \{a, b\}^\star \mid u \text{ Palindrom}\}$ ist nicht regulär, denn:

   Zu $l \geqslant 1$ wähle $w = a^l b a^l$. Sei eine Zerlegung $w = u_1 v u_2$ mit $|v| \geqslant 1$, $|u_1 v| \leqslant l$ gegeben. Dann ist uv ein Präfix des Präfixes a^l von w; damit ändert Hinzufügen von Kopien von v oder Weglassen von v die Anzahl as in diesem Präfix, lässt aber das Ende von w ab dem b unverändert, so dass die Palindromeigenschaft verlorengeht.

6.3 Sprachen als Kodaten

(Um diesen Abschnitt unabhängig von den vorhergehenden zu machen, wiederholen wir an dieser Stelle die Erinnerung an deterministische Automaten.)

Man erinnere sich, dass ein *deterministischer endlicher Automat (DFA)* $A = (Q, \Sigma, \delta, F, s)$ aus folgenden Komponenten besteht:

- einer endlichen Menge Q von *Zuständen*;
- einem endlichen Menge Σ von *Buchstaben*, dem *Alphabet*;
- einer *Transitionsabbildung* $\delta : Q \times \Sigma \to Q$;
- einem *Anfangszustand* $s \in Q$; und
- einer Menge $F \subseteq Q$ von *akzeptierenden* oder *finalen* Zuständen.

Wir sehen für unsere Zwecke das Alphabet Σ eher als Parameter des Automatenbegriffs an, und bezeichnen das verkürzte Tupel (Q, δ, F, s) als einen Σ-*Automaten*.

Die Transitionsfunktion wird äquivelenterweise repräsentiert durch Funktionen $\delta_a \colon Q \to Q$ für $a \in \Sigma$, definiert durch

$$\delta_a(q) = \delta(q, a).$$

Ein *Wort* ist ein Element von $\Sigma^\star$, also eine endliche, möglicherweise leere, Folge von Buchstaben. Wir definieren Transitionsfunktionen $\delta_w \colon Q \to Q$ für Wörter $w \in \Sigma^\star$ per

$$\begin{aligned} \delta_\epsilon(q) &= q \\ \delta_{aw}(q) &= \delta_w(\delta(a), q). \end{aligned}$$

Ein Wort w wird von A *akzeptiert*, wenn

$$\delta_w(s) \in F;$$

die Menge $L(A)$ der von A akzeptierten Wörter ist die *von A akzeptierte Sprache*.

Wir beobachten nun, dass ein DFA A aus einem sogenannten *(deterministischen Σ-)Transitionssystem* $A_0 = (Q, \delta, F)$ und einem Zustand $s \in Q$ besteht. Wir schreiben nun kurz X^Σ für das $|\Sigma|$-fache kartesische Produkt $X^{|\Sigma|} = X \times \cdots \times X$, und verwenden die $a \in \Sigma$ als Indizes für die Komponenten eines Tupels $u \in X^\Sigma$. Mittels der δ_a können wir δ als eine Abbildung

$$\langle \delta_a \rangle_{a \in \Sigma} \colon Q \to Q^\Sigma$$

repräsentieren, für die wir wieder δ schreiben (d.h. $\delta(q)$ ist das Σ-Tupel, dessen a-te Komponente $\delta_a(q)$ ist, für $a \in \Sigma$). Ferner fassen wir F als eine Abbildung $F : Q \to 2 = \{\bot, \top\}$ auf. Somit hat A_0 die Form

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

d.h. Σ-Transitionssysteme sind G-Koalgebren für

$$GX = 2 \times X^\Sigma.$$

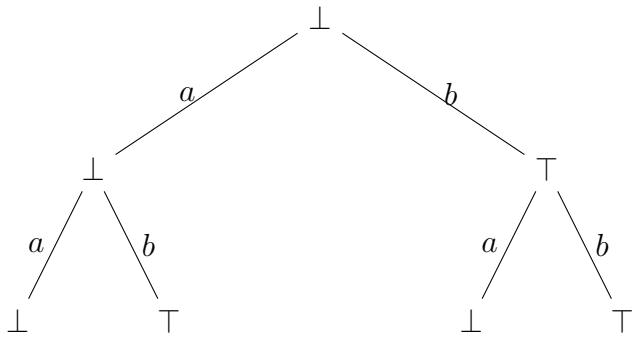
Die entsprechende Kodatentypdeklaration notieren wir wieder in Curry-Schreibweise:

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

Dies definiert die finale G-Koalgebra, deren Elemente wie folgt beschrieben sind:

- Die Elemente sind unendliche Bäume;
- die Knoten sind in *Bool* gelabelt;
- jeder Knoten hat je ein Kind für jedes $a \in \Sigma$

Für $\Sigma = \{a, b\}$ sehen die ersten drei Ebenen so eines Baums also z.B. wie folgt aus:



Wir können die Knoten eines derartigen Baums mittels *Adressen* in $\Sigma^\star$ beschreiben. Der zweite Knoten der dritten Ebene oben hat beispielsweise die Adresse ab. Der Baum ist dann vollständig beschrieben durch die Menge

$$\{u \in \Sigma^\star \mid \text{Knoten } u \text{ hat Label } \top\};$$

d.h. ein Baum ist einfach eine Sprache. Die finale G-Koalgebrastruktur $\langle F, \delta \rangle$ übersetzt sich auf Sprachen wie folgt:

- Nach der Beschreibung der finalen Koalgebrastruktur auf unendlichen Bäumen gilt $F(L) = \top$ genau dann, wenn die Wurzel des entsprechenden Baums den Label $\top$ hat; da die Wurzel durch das leere Wort ϵ adressiert wird, ist dies genau dann der Fall, wenn $\epsilon \in L$.
- Nach der Beschreibung der finalen Koalgebrastruktur auf unendlichen Bäumen ist $\delta_a(L)$ das a-te Kind der Wurzel des Baums L. Die Adresse u in diesem Teilbaum adressiert den Knoten au im ursprünglichen Baum, d.h. wir haben

  $$\delta_a(L) = \{u \in \Sigma^\star \mid au \in L\}.$$

  Somit fängt δ eine bekannte Konstruktion auf Sprachen ein, die *Ableitung* L_a einer Sprache L nach a. Für Wörter $u \in \Sigma^\star$ schreiben wir $L_u = \delta_u(L)$.

Wir bezeichnen im folgenden mit Λ sowohl die Menge aller Sprachen als auch die finale G-Koalgebra.

Nachdem wir Transitionssysteme mit G-Koalgebren identifiziert haben, erhalten wir einen natürlichen Begriff von *Morphismen* von Transitionssystemen, eben in Gestalt von Morphismen von G-Koalgebren. Für Σ-Transitionssysteme $A = (Q^A, \delta^A, F^A)$ und $B = (Q^B, \delta^B, F^B)$ ist eine Abbildung $f : Q^A \to Q^B$ demnach ein Morphismus $A \to B$, wenn das Diagramm

$$\begin{array}{ccc} Q^A & \xrightarrow{\quad f \quad} & Q^B \\ {\scriptstyle \langle F^A, \delta^A \rangle}\downarrow & \# & \downarrow{\scriptstyle \langle F^B, \delta^B \rangle} \\ 2 \times (Q^A)^\Sigma & \xrightarrow[\quad 2 \times f^\Sigma \quad]{} & 2 \times (Q^B)^\Sigma \end{array}$$

kommutiert (man erinnere sich, dass nach unserer Konvention 2 auch die Identität auf 2 bezeichnet). Da das Ergebnis beider Rechenpfade (in der rechten unteren Ecke) ein Paar ist, läuft dies auf zwei Gleichungen hinaus:

1. Kommutieren in der linken Komponente:

   $$F^B\big(f(x)\big) = F^A(x)$$

   d.h. *f lässt Finalität invariant* in dem Sinne, dass x genau dann final ist, wenn $f(x)$ final ist.

2. Kommutieren in der a-ten Komponente rechts, für $a \in \Sigma$, bedeutet, dass

   $$\delta^B_a(f(q)) = f(\delta^A_a(q)),$$

   d.h. f ist *verträglich mit Transitionen.*

Mit Blick auf korekursive Definitionen von Sprachen beschreiben wir als Nächstes die (eindeutigen) Morphismen in die finale G-Koalgebra Λ:

Satz 6.12. *Zu $A = (Q, \delta^A, F^A)$ ist*

$$\begin{aligned} f : Q &\to \Lambda \\ q &\mapsto L(A_0, q) \end{aligned}$$

der eindeutig bestimmte Transitionssystemmorphismus $A \to \Lambda$. Dabei bezeichnen wir mit (A, q) den Automaten, der aus A durch Hinzunahme von q als initialen Zustand entsteht.

In Worten: Der eindeutige Morphismus $A \to \Lambda$ bildet jeden Zustand auf seine akzeptierte Sprache ab.

Beweis (Satz 6.12). Da wir schon wissen, dass ein eindeutiger Transitionssystemmorphismus $A \to \Lambda$ existiert, reicht es zu zeigen, dass f ein solcher ist.

f lässt Finalität invariant: Ein Zustand $q \in Q$ ist offenbar genau dann final, wenn $\epsilon \in L(A_0, q)$, was nach der Beschreibung der Struktur von Λ wiederum gerade bedeutet, dass $L(A_0, q) = f(q)$ final ist.

f ist verträglich mit Transitionen, d.h. $f(\delta_a^A(q)) = \delta_a(f(q))$: Für $u \in \Sigma^*$ haben wir

$$\begin{aligned} u \in f(\delta_a^A(q)) &= L(A_0, \delta_a^A(q)) \\ &\iff \delta_u^A(\delta_a^A(q)) = \delta_{au}^A(q) \in F \\ &\iff au \in L(A_0, q) = f(q) \\ &\iff u \in \delta_a(f(q)). \end{aligned}$$

□

Korollar 6.13. *1. $L \in \Lambda$ akzeptiert L*

2. Morphismen von Transitionssystemen bewahren die akzeptierte Sprache; d.h. wenn $f : A_0 \to B_0$ ein Morphismus von Transitionssystemen ist, gilt $L(A_0, q) = L(B_0, f(q))$.

Beweis. Nach Satz 6.12 ist die von einem Zustand akzeptierte Sprache sein Verhalten im Sinne von Bemerkung 4.39. □

6.4 Minimierung

Wir entwickeln nun eine koalgebraische Sicht auf die Minimierung von deterministischen Automaten, d.h. die Überführung eines gegebenen Automaten einen äquivalenten Automaten (d.h. einen, der dieselbe Sprache akzeptiert) mit minimaler Anzahl Zustände.

Definition 6.14. Für einen Zustand q in einem Transitionssystem A definieren wir das von q *erzeugte Untertransitionssystem* $\langle q \rangle$ durch

$$\langle q \rangle = \{\delta_u(q) \mid u \in \Sigma^\star\}.$$

Es ist klar, dass $\langle q \rangle$ abgeschlossen unter δ ist; somit ist $\langle q \rangle$ mit der von A geerbten Transitionsabbildung tatsächlich ein Transitionssystem.

Lemma 6.15. *Es gilt $L(\langle q \rangle, q) = L(A, q)$.*

Beweis. Die Einbettung $\langle q \rangle \hookrightarrow A$ ist ein Morphismus, so dass die Behauptung aus Korollar 6.13 folgt. □

Damit folgt, wiederum per Korollar 6.13,

Lemma 6.16. *Für jedes Sprache $L \in \Lambda$ gilt*

$$L(\langle L \rangle, L) = L.$$

Wir haben also mit $(\langle L \rangle, L)$ einen (eventuell unendlichen) Automaten gefunden, der L akzektiert. Wir werden zeigen, dass dies der kleinste solche Automat ist. Wir verwenden folgende einfache Tatsache:

Lemma 6.17. *Jeder Morphismus f von Transitionssystemen schränkt ein zu einer Surjektion*

$$f : \langle q\rangle \twoheadrightarrow \langle f(q)\rangle.$$

Insbesondere gilt $|\langle f(q)\rangle| \leqslant \langle q\rangle$.

(Dabei bezeichnet $|A|$ die Anzahl Zustände von A.)

Beweis. Man erinnere sich, dass $\langle q\rangle = \{\delta_u(q) \mid u \in \Sigma^\star\}$ und $\langle f(q)\rangle = \{\delta_u(f(q)) \mid u \in \Sigma^\star\}$. Die Behauptung folgt also aus $f(\delta_u(q)) = \delta_u(f(q))$ (für $u \in \Sigma^\star$), was sich wiederum leicht per Induktion über u aus der Morphismeneigenschaft von f ergibt. □

Da, wie oben gezeigt, die Abbildung $L(A, -)$ ein Morphismus von Transitionssystemen ist, gilt also insbesondere für jedes Transitionssystem A und jeden Zustand q in A

$$|\langle L(A, q)\rangle| \leqslant |\langle q\rangle| \leqslant |A|$$

Wenn also eine Sprache von einem Automaten (A, q) akzeptiert wird, also $L(A, q) = L$, dann ist $|A| \geqslant |\langle L\rangle|$, das heisst, $\langle L\rangle$ ist der *minimale Automat* für L. Dies zeigt insbesondere:

Satz 6.18. *Eine Sprache L ist genau dann regulär, wenn $\langle L\rangle$ endlich ist.*

Wir erinnern nunmehr an einige mengentheoretische Konzepte. Gegeben eine Äquivalenzrelation R auf einer Menge X ist die *Äquivalenzklasse* $[x]_R$ von $x \in X$ definiert als $[x]_R = \{y \in X \mid xRy\}$. Die *Quotientenmenge* X/R ist dann gegeben als $X/R = \{[x]_R \mid x \in X\}$. Wir sagen, dass R *endlichen Index* hat, wenn X/R endlich ist. Wenn nun $f : X \to Y$ eine Abbildung ist, definiert f auf X eine Äquivalenzrelation $\operatorname{Ker} f$ durch $x(\operatorname{Ker} f)y \iff f(x) = f(y)$. Wir haben dann eine bijektive Abbildung

$$h : X/\operatorname{Ker} f \to f[X],$$

definiert durch $h([x]_R) = f(x)$. Insbesondere hat also das Bild $f[X]$ von f dieselbe Kardinalität wie $X/\operatorname{Ker} f$. Ende der Erinnerung.

Nun ist $\langle L\rangle = \{L_u \mid u \in \Sigma^\star\}$ gerade das Bild der Abbildung $d : \Sigma^\star \to \Lambda$ mit $d(u) = L_u$, ist also in Bijektion mit $\Sigma^\star/\operatorname{Ker} d$. Wir schreiben $\sim_L := \operatorname{Ker} d$; dann gilt für $v, w \in \Sigma^\star$

$$\begin{aligned} v \sim_L w &\iff L_v = L_w \\ &\iff \forall u \in \Sigma^\star.\, (u \in L_v \Leftrightarrow u \in L_w) \\ &\iff \forall u \in \Sigma^\star.\, (vu \in L \Leftrightarrow wu \in L). \end{aligned}$$

Damit ist also Satz 6.18 gerade der klassische Satz von Myhill und Nerode, den wir noch einmal ausdrücklich formulieren:

Satz 6.19 (Myhill und Nerode)**.** *Eine Sprache L ist genau dann regulär, wenn die durch*

$$v \sim_L w \iff \forall u \in \Sigma^\star.\, (vu \in L \Leftrightarrow wu \in L)$$

definierte Äquivalenzrelation $\sim_L$ endlichen Index hat.

Wir können mit den nun angesammelten Mitteln auch leicht einen Minimierungsalgorithmus für DFA angeben:

Algorithmus 6.20. (Minimiere einen DFA $A = (Q, \Sigma, \delta, s, F)$) Der Algorithmus verwendet eine globale Variable $R \subseteq Q \times Q$. Er läuft wie folgt:

1. Entferne aus Q alle nicht erreichbaren Zustände.
2. Initialisiere R auf $\{(q_1, q_2) \mid q_1 \in F \iff q_2 \in F\}$
3. Suche ein Paar $(q_1, q_2) \in R$ und einen Buchstaben $a \in \Sigma$ mit

   $$(\delta_a(q_1), \delta_a(q_2)) \notin R.$$

   Wenn kein solches Paar gefunden wird, gehe zu Schritt 4. Andernfalls entferne (q_1, q_2) und (q_2, q_1) aus R und fahre bei 3. fort.
4. Identifiziere alle Zustandspaare in R.

Wenn man den Algorithmus von Hand verwendet, sollte man ausnutzen, dass offenbar R stets symmetrisch ist und alle Paare (q, q) stets in R verbleiben. Es reicht also, eine dreieckige Tabelle zu führen, in der jedes Paar nur in einer der beiden möglichen Anordnungen vorkommt, und die Diagonalelemente (q, q) gar nicht. Man streicht dann zunächst die Zustandspaare, die hinsichtlich Finalität nicht übereinstimmen, und streicht dann wiederholt Paare heraus, die nach dem Kriterium in Schritt 3 aus R zu entfernen sind (dabei ist aus der dreieckigen Tabelle natürlich effektiv jeweils nur ein Paar zu entfernen, die Entfernung des umgekehrten Paares ist dann implizit).

Korrektheit des Algorithmus halten wir wie folgt fest:

Satz 6.21. *Nach Beendigung des Minimierungsalgorithmus ist R eine Äquivalenzrelation, und $A/R = (Q/R, \delta^R, F/R, [s]_R)$ mit $\delta_a^R([q]_R) = [\delta_a(q)]_R$ ist der kleinste Σ-Automat, der $L(A)$ akzeptiert.*

(Wir bezeichnen hier Äquivalenzklassen unter R mit $[q]_R = \{q' \mid qRq'\}$ und erinnern erneut daran, dass $Q/R = \{[q]_R \mid q \in Q\}$ die Quotientenmenge modulo R ist.)

Zum Beweis erweitern wir noch den Begriff der Bisimulation auf beliebige Transitionssysteme (statt nur das finale Transitionssystem):

Definition 6.22. Sei $A = (Q, \delta, F)$ ein Transitionssystem. Eine Relation $R \subseteq Q \times Q$ ist eine *Bisimulation*, wenn für qRr und $a \in \Sigma$ stets

1. $q \in F \iff r \in F$ und
2. $\delta_a(q)\, R\, \delta_a(r)$

gilt. Zwei Zustände p, q sind *bisimilar*, wenn es eine Bisimulation R mit pRq gibt.

Lemma 6.23. *Zwei Zustände in einem Transitionssystem sind genau dann bisimilar, wenn sie dieselbe Sprache akzeptieren.*

Beweis. „⇒“: Sei R eine Bisimulation auf einem Transitionssystem A. Mittels der Tatsache, dass $L(A,-)$ ein Transitionssystemmorphismus ist, prüft man leicht, dass dann die Relation

$$\{(L(A,q), L(A,r)) \mid (q,r) \in R\}$$

eine Bisimulation auf dem finalen Transitionssystem ist, und damit enthalten in der Identitätsrelation; d.h. aus $q\,R\,r$ folgt $L(A,r) = L(A,r)$.

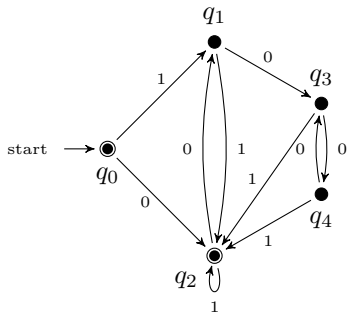
„⇐“: Auf jedem Transitionssystem A ist die durch

$$q\,R\,r \iff L(A,q) = L(A,r)$$

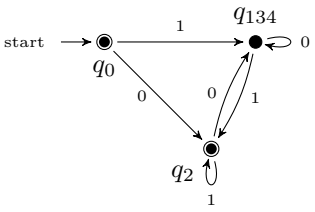
definierte Relation eine Bisimulation. □

Beweis (Satz 6.21). Nach den obigen Betrachtungen reicht es zu zeigen, dass zwei Zustände q, r in A genau dann bisimilar sind, wenn Sie bis zum Ende in R verbleiben. Wir entfernen aus R klarerweise nur Zustandspaare, die *nicht* bisimilar sind; es reicht also, zu zeigen, dass R am Ende eine Bisimulation ist. Das ist aber klar anhand der Abbruchbedingung. Damit ist dann auch die Abbildung δ^R durch die angegebene Vorschrift wohldefiniert, d.h. $[\delta(a,q)]_R$ hängt nicht von der Wahl des Repräsentanten der Äquivalenzklasse $[q]_R$ ab. □

Beispiel 6.24. Wir minimieren den Automaten links in der folgenden Grafik. In der Mitte die bereits ausgefüllte Tabelle. An den Zahlen lässt sich die Iteration ablesen, in der das jeweilige Feld markiert wurde. Rechts der fertig reduzierte Automat.



q_1	0			
q_2	1	0		
q_3	0		0	
q_4	0		0	
	q_0	q_1	q_2	q_3



6.5 Reguläre Ausdrücke per Korekursion

Nach den Resultaten der vorigen Abschnitte können wir die Semantik regulärer Ausdrücke alternativ definieren, indem wir ein Transitionssystem $\mathcal{E} \to 2 \times \mathcal{E}^\Sigma$ auf der Menge $\mathcal{E}$ der regulären Ausdrücke angeben. Letzteres definieren wir wiederum *rekursiv*. Wir erinnern an die Definition regulärer Ausdrücke in Form einer Datentypdeklaration:

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

Wir definieren dann $\langle F, \delta\rangle : \mathcal{E} \to 2 \times \mathcal{E}^\Sigma$ rekursiv wie folgt:

- $F(0) = \bot$ und
  $\delta_a(0) = 0$
- $F(1) = \top$ und
  $\delta_a(1) = 0$
- $F(a) = \bot$ und
  $\delta_a(a) = 1$, $\delta_b(a) = 0$ für $b \neq a$
- $F(r + s) = F(r) \vee F(s)$ und
  $\delta_a(r + s) = \delta_a(r) + \delta_a(s)$
- $F(rs) = F(r) \wedge F(s)$ und
  $\delta_a(rs) = \begin{cases} \delta_a(r)s & \text{falls } F(r) = \bot \\ \delta_a(r)s + \delta_a(s) & \text{sonst} \end{cases}$
- $F(r^\star) = \top$ und
  $\delta_a(r^\star) = \delta_a(r)r^\star$

Das definiert eine Abbildung $L : \mathcal{E} \to \Lambda$ korekursiv per

- $F(L(r)) \Leftrightarrow F(r)$ (also $\epsilon \in L(r) \Leftrightarrow F(r)$)
- $\delta_a(L(r))(= L(r)_a) = L(\delta_a(r))$

Unter Verwendung dieser korekursiven Definition können wir nunmehr koinduktive Beweise der Äquivalenz regulärer Ausdrücke führen. Wir erinnern uns dazu, dass eine Relation $R \subseteq \Lambda \times \Lambda$ eine Bisimulation ist, wenn für alle $L\,R\,K \in \Lambda$ gilt:

- $F(L) = F(K)$
- $L_a\,R\,K_a$ für alle $a \in \Sigma$

Beispiel 6.25 (Gleichheit (der erzeugten Sprachen) von regulären Ausdrücken)**.** Wir verwechseln der Kürze halber reguläre Ausdrücke mit den durch sie definierten formalen Sprachen. Insbesondere schreiben wir für $r \in \mathcal{E}$ kurz r_a statt $\delta_a(r)$.

1. Wir zeigen $r + 0 = r$ koinduktiv, indem wir zeigen, dass

   $$R = \{(r + 0, r) \mid r \in \mathcal{E}\}$$

   eine Bisimulation ist:

   - $F(r + 0) = F(r) \vee F(0) = F(r) \vee \bot = F(r)$
   - $(r + 0)_a = r_a + 0_a = r_a + 0\,R\,r_a$

2. Wir zeigen $r(s + t) = rs + rt$ koinduktiv.

   Wir versuchen zunächst zu zeigen, dass $R = \{(r(s + t), rs + rt) \mid r, s, t \in \mathcal{E}\}$ eine Bisimulation ist: Wir haben, im etwas komplizierteren Fall $F(r) = \top$,

$$(r\ (s+t))_a = (r_a\ (s+t) + s + t)_a = r_a\ (s+t) + s_a + t_a$$

und

$$(rs+rt)_a = (rs)_a + (rt)_a = r_a s + s_a + r_a t + t_a = r_a s + r_a t + s_a + t_a,$$

so dass die beiden Seiten nur bis auf Kongruenz bezüglich + in Relation stehen, wobei wir hier so tun, als hätten wir Assoziativität und Kommutativität von + schon bewiesen. Wir wählen daher stattdessen allgemeiner

$$R = \{(r(s+t)+p, rs+rt+p) \mid r,s,t,p \in \mathcal{E}\};$$

man beachte erneut die Analogie zur Strategie der Verstärkung von Induktionsbehauptungen. Damit ist R nun tatsächlich eine Bisimulation:

- Wir haben

  $$F(r(s+t+p)) = (F(r) \wedge (F(s) \vee F(t))) \vee F(p)$$

  und

  $$F(rs+rt+p) = (F(r) \wedge F(s)) \vee (F(r) \wedge F(t)) \vee F(p),$$

  was nach dem Distributivgesetz der Aussagenlogik äquivalent ist.
- Wir schränken uns wieder auf den komplizierteren Fall $F(r) = \top$ ein, und haben dann

  $$\begin{aligned}(r(s+t)+p)_a &= s_a + t_a + r_a(s+t) + p_a \\ &\ R\ s_a + t_a + r_a s + r_a t + p_a \\ &= (rs+rt+p)_a.\end{aligned}$$

3. Seien r, s, t reguläre Ausdrücke mit

   $$F(r) = \bot \quad \text{und} \quad s = rs + t.$$

   Wir zeigen koinduktiv, dass dann

   $$s = r^* t.$$

   (Ohne die Annahme $F(r) = \bot$ folgt dies im Allgemeinen nicht, z.B. betrachte man den Fall $r = 1$, $t = 0$.)

   Wir versuchen es wie üblich zunächst mit der direkt aus der Behauptung gewonnenen Relation $R = \{(s, r^*t)\}$. Für $a \in \Sigma$ rechnen wir wie folgt:

   $$\begin{aligned}s_a &= (rs+t)_a && \text{(Annahme)} \\ &= (rs)_a + t_a \\ &= r_a s + t_a && (F(r) = \bot)\end{aligned}$$

sowie

$$\begin{aligned}(r^*t)_a &= (r^*)_a t + t_a && (F(r^*) = \top)\\ &r_a r^* t + t_a.\end{aligned}$$

Wir finden hier also zwar jeweils s und r^*t wieder, aber in einem Kontext der Form $u(\cdot) + p$. Wir setzen letztlich also statt unseres obigen Versuchs

$$R = \{(us + p, ur^*t + p) \mid u, p \in \mathcal{E}\}.$$

Dies ist nun tatsächlich eine Bisimulation:

- Wir haben
$$F(us + p) = (F(u) \wedge F(s)) \vee F(p)$$
und
$$F(ur^*t + p) \;=\; (F(u) \wedge F(r^*) \wedge F(t)) \vee F(p) \;=\; (F(u) \wedge F(t)) \vee F(p),$$
was deswegen gleich ist, weil nach Annahme
$$F(s) = F(rs + t) = (F(r) \wedge F(s)) \vee F(t) = F(t).$$

- Für $a \in \Sigma$ haben wir im komplizierteren Fall $F(u) = \top$ (der Fall $F(u) = \bot$ ist ähnlich, aber einfacher)
$$\begin{aligned}(us + p)_a &= u_a s + s_a + p_a\\ &= u_a s + (rs + t)_a + p_a\\ &= u_a s + r_a s + t_a + p_a && (F(r) = \bot)\\ &= (u_a + r_a)s + t_a + p_a,\end{aligned}$$
wobei wir im letzten Schritt das oben bewiesene Distributivgesetz verwenden, sowie
$$\begin{aligned}(u(r^*t) + p)_a &= u_a r^* t + (r^*t)_a + p_a\\ &= u_a r^* t + r_a^* t + t_a + p_a\\ &= u_a r^* t + r_a r^* t + t_a + p_a\\ &= (u_a + r_a) r^* t + t_a + p_a,\end{aligned}$$
wiederum unter Verwendung des Distributivgesetzes im letzten Schritt; wie verlangt stehen die beiden Seiten also in Relation R.

Damit folgt dann wie behauptet $s = r^*t$ (warum?).