4

Term

In diesem Kapitel geht es um Regeln zum Ersetzen in Bäumen. Bäume werden dabei oft als spezielle Wörter geschrieben, nämlich als *Terme* über Funktionssymbolen, Konstanten und Variablen, oder als logische Formeln über Prädikatsymbolen. Für das Verständnis dieser Ersetzungssysteme ist es aber wichtig zu sehen, dass Formeln und Terme eigentlich Bäume sind, denn diese Tatsache bestimmt die Eigenschaften dieser Systeme.

Zum Inhalt dieses Kapitels werden mehrere Texte ins Netz gestellt, so dass hier nur eine kurze Zusammenfassung der Inhalte steht, mit einigen Beispielen und inhaltlichen Ergänzungen.

4.1 Termersetzung

Dieser Abschnitt orientiert sich an [KdV03], auch in der Notation, behandelt aber längst nicht alles dort Geschriebene. Oft wird nur auf diesen Text verwiesen; alle abweichende Definitionen oder Notationen werden aber hier aufgeschrieben.

4.1.1 Terme

Terme werden als *Wörter* über Funktionssymbolen und Variablen definiert. Dies ist ein Widerspruch zu unserer Behauptung, Terme seien Bäume, wird sich aber gleich aufklären.

Sei Σ eine endlich Menge von *Funktionssymbolen*, und X eine Menge von *Termvariablen*, die aus technischen Gründen auch unendlich viele Symbole enthalten kann. Σ zerfällt in wechselseitig disjunkte Teilmengen Σ_k von Funktionssymbolen der *Stelligkeit* $k \geqslant 0$. Die nullstelligen Funktionssymbole aus Σ_0 heißen *Konstantensymbole*. Variablen haben ebenfalls die Stelligkeit 0.

Definition 4.1 (Term). Die *Terme* über Σ und X ist die kleinste Menge von Wörtern $\mathcal{T}(X) \subseteq (\Sigma \cup X)^*$, die folgende Regeln erfüllt:

$$\frac{x \in X}{x \in \mathcal{T}(X)} \qquad \frac{c \in \Sigma_0}{c \in \mathcal{T}(X)} \qquad \frac{f \in \Sigma_k,\ t_1, \ldots, t_k \in \mathcal{T}(X),\ k > 0}{f\, t_1 \cdots t_k \in \mathcal{T}(X)}$$

$\mathcal{T}$ ist die Menge der *Grundterme*, die keine Variablen enthalten, sondern ohne die erste Regel gebildet werden.

Definition 4.2 (Variablenmenge). Für Terme $t \in \mathcal{T}(X)$ bezeichnet $Var(t) \subseteq X$ die *Variablenmenge* von t, die rekursiv über die Struktur von Termen definiert wird als:

$$\begin{aligned} Var(x) &= \{x\} && \text{für } x \in X \\ Var(c) &= \{\} && \text{für } c \in \Sigma_0 \\ Var(f\, t_1 \cdots t_k) &= \textstyle\bigcup_{i=1}^{n} Var(t_i) && \text{für } f \in \Sigma_k, t_i \in \mathcal{T}(X), 1 \leqslant i \leqslant k, k > 0 \end{aligned}$$

Variablen sind Platzhalter für Terme.

Definition 4.3 (Substitution). Eine Abbildung $\sigma\colon X \to \mathcal{T}(X)$ wird (*Variablen-*) *Substitution* genannt. Die Anwendung einer Substitution σ auf einen Term $t \in \mathcal{T}(X)$ wird (*Term-*) *Substitution* genannt, als t^σ geschrieben und ist rekursiv über die Termstruktur definiert als

$$\begin{aligned} x^\sigma &= \sigma(x) && \text{für } x \in X \\ c^\sigma &= c && \text{für } c \in \Sigma_0 \\ (f\, t_1 \cdots t_k)^\sigma &= f\, t_1^\sigma \cdots t_k^\sigma && \text{für } f \in \Sigma_k, t_i \in \mathcal{T}(X), 1 \leqslant i \leqslant k, k > 0 \end{aligned}$$

Definition 4.4 (Positionen,Unterterm, Untertermersetzung). Die *Positionen* eines Terms $t \in \mathcal{T}(X)$ sind Wörter $\Delta(t) \subseteq \mathbb{N}_+^*$ aus positiven Zahlen:

$$\begin{aligned} \Delta(x) = \{\Box\} \text{ und } \Delta(c) = \{\Box\} && && \text{für } x \in X \text{und } c \in \Sigma_0 \\ \Delta(f\, t_1 \cdots t_k) \;=\; \{\Box\} \cup \{i\,\omega \mid 1 \leqslant i \leqslant k, \omega \in \Delta(t_i)\} && && \text{für } f \in \Sigma_k, t_i \in \mathcal{T}(X) \\ && && 1 \leqslant i \leqslant k, k > 0 \end{aligned}$$

Für jede Position $\omega \in \Delta(t)$ definiert t/ω den *Unterterm* an Position ω wie folgt:

$$\begin{aligned} t/\Box &= t \\ (f\, t_1 \cdots t_k)/i\,\omega &= t_i/\omega \end{aligned}$$

In einem Term wird die *Ersetzung* des Unterterms t/ω durch einen Term u geschrieben als $t[\omega \leftarrow u]$ und so definert:

$$\begin{aligned} t[\Box \leftarrow u] &= u \\ (f\, t_1 \cdots t_k)[i\,\omega \leftarrow u] &= f\, t_1 \cdots t_{i-1}\, (t_i[\omega \leftarrow u])\, t_{i+1} \cdots t_k \end{aligned}$$

Beispiel 4.1 (Terme). Nehmen wir an, die Signatur Σ enthielte $\Sigma_0 = \{0, \ldots\}$, $\Sigma_1 = \{\mathrm{S}, \ldots\}$ und $\Sigma_2 = \{\mathrm{A}, \mathrm{M}, \ldots\}$, und $X = \{x, y, z, \ldots\}$.

Dann ist $\mathrm{A}\, x\, \mathrm{M}\, y\, y \in \mathcal{T}(X)$ mit der Variablenmenge

$$Var(\mathrm{A}\, x\, \mathrm{M}\, y\, x) = Var(x) \cup Var(\mathrm{M}\, y\, x) = \{x\} \cup Var(x) \cup Var(y) = \{x, y\}$$

und den Positionen

$$\begin{aligned}\Delta(\mathrm{A}\ x\ \mathrm{M}\ y\ x) &= \{\Box\} \cup \{1\,\omega_1 \mid \omega_1 \in \Delta(x)\} \cup \{2\,\omega_2 \mid \omega_2 \in \Delta(\mathrm{M}\ y\ x)\} \\ &= \{\Box\} \cup \{1\,\Box\} \cup \{2\,\omega_2 \mid \omega_2 \in \Delta(\mathrm{M}\ y\ x)\} \\ &= \{\Box, 1\} \cup \{2\,\omega_2 \mid \omega_2 \in \Delta(\mathrm{M}\ y\ x)\}\end{aligned}$$

Dabei ist

$$\begin{aligned}\Delta(\mathrm{M}\ y\ x) &= \{\Box\} \cup \{1\,\omega_3 \mid \omega_3 \in \Delta(x)\} \cup \{2\,\omega_4 \mid \omega_4 \in \Delta(y)\} \\ &= \{\Box\} \cup \{1\,\Box\} \cup \{2\,\Box\} \\ &= \{\Box, 1, 2\}\end{aligned}$$

Das ergibt $\Delta(\mathrm{A}\ x\ \mathrm{M}\ y\ x) = \{\Box, 1, 2, 21, 22\}$ und wir können die folgenden Unterterme auswählen:

$$\begin{aligned}(\mathrm{A}\ x\ \mathrm{M}\ y\ x)/\Box &= \mathrm{A}\ x\ \mathrm{M}\ y\ x \\ (\mathrm{A}\ x\ \mathrm{M}\ y\ x)/1 &= x & (\mathrm{A}\ x\ \mathrm{M}\ y\ x)/2 &= \mathrm{M}\ y\ x \\ (\mathrm{A}\ x\ \mathrm{M}\ y\ x)/12 &= y & (\mathrm{A}\ x\ \mathrm{M}\ y\ x)/22 &= x\end{aligned}$$

Eine Variablensubstitution σ mit $\sigma(x) = \mathrm{S\,A\,0}\,x$ und $\sigma(y) = x$ kann auf diesen Term angewendet werden:

$$\begin{aligned}(\mathrm{A}\ x\ \mathrm{M}\ y\ x)^\sigma &= \mathrm{A}\ x^\sigma\ (\mathrm{M}\ y\ x)^\sigma \\ &= \mathrm{A}\ \sigma(x)\ \mathrm{M}\ y^\sigma\ x^\sigma \\ &= \mathrm{A\,S\,A\,0}\ x\ \mathrm{M}\ \sigma(y)\ \sigma(x) \\ &= \mathrm{A\,S\,A\,0}\ x\ \mathrm{M}\ x\ \mathrm{S\,A\,0}\ x\end{aligned}$$

Den gleichen Term liefert die Termersetzung

$$\begin{aligned}&(\mathrm{A}\ x\ \mathrm{M}\ y\ x)[1 \leftarrow \mathrm{S\,A\,0}\ x]\ [21 \leftarrow x]\ [22 \leftarrow \mathrm{S\,A\,0}\ x] \\ =\ &(\mathrm{A}\ x[\Box \leftarrow \mathrm{S\,A\,0}\ x]\ \mathrm{M}\ y\ x)\ [21 \leftarrow x]\ [22 \leftarrow \mathrm{S\,A\,0}\ x] \\ =\ &(\mathrm{A\,S\,A\,0}\ x\ \mathrm{M}\ y\ x)\ [21 \leftarrow x]\ [22 \leftarrow \mathrm{S\,A\,0}\ x] \\ =\ &\mathrm{A\,S\,A\,0}\ x\ (\mathrm{M}\ y\ x)[1 \leftarrow x]\ [22 \leftarrow \mathrm{S\,A\,0}\ x] \\ =\ &\mathrm{A\,S\,A\,0}\ x\ \mathrm{M}\ (y)[\Box \leftarrow x]\ x\ [22 \leftarrow \mathrm{S\,A\,0}\ x] \\ =\ &\mathrm{A\,S\,A\,0}\ x\ \mathrm{M}\ x\ x\ [22 \leftarrow \mathrm{S\,A\,0}\ x] \\ =\ &\mathrm{A\,S\,A\,0}\ x\ (\mathrm{M}\ x\ x)\ [2 \leftarrow \mathrm{S\,A\,0}\ x] \\ =\ &\mathrm{A\,S\,A\,0}\ x\ \mathrm{M}\ x\ (x)\ [\Box \leftarrow \mathrm{S\,A\,0}\ x] \\ =\ &\mathrm{A\,S\,A\,0}\ x\ \mathrm{M}\ x\ \mathrm{S\,A\,0}\ x\end{aligned}$$

4.1.2 Ersetzungsregeln

Definition 4.5 (Termersetzungsregel). Ein Paar von Termen $l, r \in \mathcal{T}(X)$ ist eine *Termersetzungsregel* und wird geschrieben als $l \to r$, wenn gilt:

1. Die linke Seite l ist keine Variable: $l \notin X$.
2. Die rechte Seite r enthält nur Variablen, die auch in der linken Seite l auftreten: $\mathit{Var}(r) \subseteq \mathit{Var}(l)$.

Eine endliche Menge R von Termersetzungsregeln nennen wir ein *Termersetzungssystem*.

Definition 4.6. Aus einem Termersetzungssystem R kann folgende *Termersetzungs-Relation* $\Rightarrow_R \subseteq \mathcal{T}(X) \times \mathcal{T}(X)$ abgeleitet werden:

$$\frac{l \to r \in R}{l \Rightarrow_R r}\ \text{(Einschluss)} \qquad \frac{t \Rightarrow_R u, t' \in \mathcal{T}(X), x \in X}{t\{x \mapsto t'\} \Rightarrow_R u\{x \mapsto t'\}}\ \text{(Substitution)}$$

$$\frac{f\,t_1 \cdots t_k \in \mathcal{T}(X), t_i \Rightarrow_R u, 1 \leqslant i \leqslant k}{f\,t_1 \cdots t_k \Rightarrow_R f\,t_1 \cdots t_{i-1}\,u\,t_{i+1} \cdots t_k}\ \text{(Einbettung)}$$

Diese Regeln definieren Ersetzungen auf Termen mit Variablen, obwohl praktisch gesehen meist nur die Ersetzung auf Grundtermen betrachtet wird. In der Terminologie der abstrakten Reduktion aus Abschnitt 2.1 bildet R also ein Reduktionssystem $\langle \mathcal{T}(X), \Rightarrow_R \rangle$ auf allgemeinen Termen; dies könnte man bei Bedarf auf ein "Teil"-Ersetzungssystem $\langle \mathcal{T}, \Rightarrow'_R \rangle$ mit einer Ersetzungsrelation $\Rightarrow'_R = \Rightarrow_R \cap\, \mathcal{T} \times \mathcal{T}$ auf Grundtermen beschränken. Für die relativ grundlegenden Eigenschaften, die wir hier darstellen, ist das aber nicht nötig.

Satz 4.1. *$t \Rightarrow_R u$ gilt genau dann wenn es eine Regel $l \to r \in R$, eine Position $\omega \in \Delta(t)$ und eine Substitution $\sigma\colon X \to \mathcal{T}(X)$ gibt, so dass*

1. *$t/\omega = l^\sigma$ und*
2. *$u = t[\omega \leftarrow r^\sigma]$.*

Beispiel 4.2 (Addition und Multiplikation natürlicher Zahlen). Für die Signatur aus Beispiel 4.1 definieren wir die Funktionssymbole $\mathrm{A}, \mathrm{M} \in \Sigma_2$ als Multiplikation und Addition. Dabei stellt ein Term der Form $S^n\,0$ die natürliche Zahl n dar.

$$\begin{aligned}\rho_1 &: \mathrm{A}\ x\ 0 \to x & \rho_2 &: \mathrm{A}\ x\ \mathrm{S}\ y \to \mathrm{S\,A}\ x\ y \\ \rho_3 &: \mathrm{M}\ x\ 0 \to 0 & \rho_4 &: \mathrm{M}\ x\ \mathrm{S}\ y \to \mathrm{A}\ x\ \mathrm{M}\ x\ y\end{aligned}$$

In Regeln wie diesen erlauben wir uns eine etwas leichter lesbare Notation für Terme: Wir schreiben die Argumente einer Funktion in Klammern und trennen sie mit Kommata, und statt mancher Funktionssymbole wie A und M benutzen wir die aus der Mathematik geläufigen Infix-Operatorsymbole "+" und "∗". Dann sehen die Regeln so aus:

$$\begin{aligned}\rho_1 &: x + 0 \to x & \rho_2 &: x + \mathrm{S}(y) \to \mathrm{S}(x + y) \\ \rho_3 &: x * 0 \to 0 & \rho_4 &: x * \mathrm{S}(y) \to x + (x * y)\end{aligned}$$

Hier eine normalisierende Ersetzungssequenz mit diesen Regeln:

$$\begin{aligned}(\mathrm{S}(0) * 0) + \mathrm{S}(0) &\Rightarrow_{\rho_2} \mathrm{S}((\mathrm{S}(0) * 0) + 0) \\ &\Rightarrow_{\rho_1} \mathrm{S}(\mathrm{S}(0) * 0) \\ &\Rightarrow_{\rho_3} \mathrm{S}(0)\end{aligned}$$

Dies ist beileibe nicht die einzige mögliche Ersetzungssequenz:

$$\begin{aligned}(\mathrm{S}(0) * 0) + \mathrm{S}(0) &\Rightarrow_{\rho_3} 0 + \mathrm{S}(0)\\ &\Rightarrow_{\rho_2} \mathrm{S}(0 + 0)\\ &\Rightarrow_{\rho_1} \mathrm{S}(0)\end{aligned}$$

In diesem Fall kommt die gleiche Normalform heraus.

Terme können als Bäume dargestellt werden, genauer gesagt: als endliche markierte Bäume mit geordneten Nachfolgern. Der Baum eines Terms t ist wie folgt definiert:

- Eine Variable oder ein Konstantensymbol wird durch einen Baum mit einem einzigen Knoten dargestellt, der mit der Variablen bzw. Konstanten markiert ist.
- Der Term $f\, t_1 \cdots t_k$ wird durch einen Baum mit einer Wurzel dargestellt, die mit f markiert ist und k direkte Unterbäume hat, die – von links nach rechts – die Terme t_1 bis t_k darstellen.

Abbildung 4.1 zeigt die Regeln des Beispiels als Baumersetzungsregeln, und Abbildung 4.3 alle Baumersetzungsschritte für den Anfangsterm der Termersetzungssequenz.

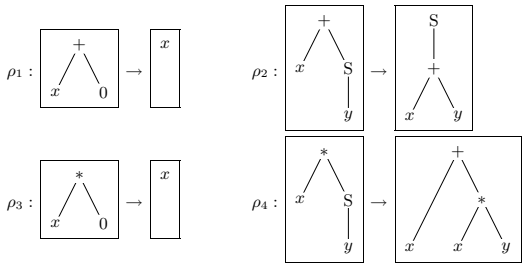


Abb. 4.1. Termbaum-Ersetzungsregeln für Addition und Multiplikation

Beispiel 4.3 (Gruppenaxiome). Gruppen werden durch folgende Gleichungen charakterisiert:

$$x + 0 = x \qquad x + (-x) = 0 \qquad (x + y) + z = x + (y + z)$$

Weshalb können diese Gleichungen nicht direkt als Ersetzungsregeln benutzt werden, die von links nach rechts oder und von rechts nach links angewendet

werden können? Die einfache Antwort lautet: Die umgekehrte Regeln sind nicht immer Termersetzungsregeln nach Definition 4.5: $x \to x + 0$ verletzt Bedingung 1, und $0 \to x + (-x)$ verletzt Bedingung 2.

Weshalb sind solche Regeln aber verboten?

1. Regeln wie $x \to x + 0$ können an jeder Stelle in einem Term angewendet werden, so dass es keine Normalformen gibt, sondern nur nicht terminierende Ersetzungen.
2. Regeln wie $0 \to x + (-x)$ führen neue Variablen ein, die beliebig substituiert werden können, so dass es keine eindeutigen Normalformen bzw. keine Konfluenz geben kann.

Ein terminierende Termersetzungssystem für die Gruppenaxiome sieht so aus:

$$\begin{aligned} x + 0 &\to x & 0 + x &\to x\\ x + (-x) &\to 0 & (-x) + x &\to 0\\ -0 &\to 0 & -(-x) &\to x\\ -(x + y) &\to (-x) + (-y) & (x + y) + z &\to x + (y + z)\\ (-x) + (x + y) &\to y & x + ((-x) + y) &\to y \end{aligned}$$

Termersetzungsregeln sind Schemata: sie enthalten Variablen, die Platzhalter für beliebige Terme sind. Sie werden benutzt, um Terme zu *reduzieren*, d.h. Regeln anzuwenden, solange das möglich ist. Terme, auf die keine Regeln angewendet werden können, heißen *Normalformen*. Betrachten wir ein Beispiel:

$$(-(x + x)) + ((x + x) + (y + (-0))) \;\Rightarrow\; y + (-0) \;\Rightarrow\; y + 0 \;\Rightarrow\; y$$

Abbildung 4.2 zeigt die Sequenz, wobei die Terme als Bäume dargestellt werden. Im ersten Ersetzungsschritt ist in der angewendeten Regel (der letzten) die Variable x Platzhalter für den term $x + x$, und y für $y + (-0)$. Der zweite Schritt ersetzt den Teilterm -0 , und der Term y ist eine Normalform. Der erste und zweite Schritt können vertauscht werden, aber das ändert am Ergebnis nichts. Das vorliegende Regelsystem ist terminierend und konfluent: Jeder Ersetzungssequenz führt zu einer Normalform, und alle Sequenzen für einen bestimmten Term errechnen immer die gleiche Normalform.

4.1.3 Termination und Konfluenz

Die von einem Ersetzungssystem definierten Funktionen sind im Allgemeinen partiell und nicht deterministisch.

Wir betrachten, unter welchen Bedingungen diese Funktionen total bzw. eindeutig sind. Dazu muss das Ersetzungssystem $\langle \mathcal{T}, \Rightarrow_R \rangle$ terminierend bzw. konfluent sein. Leider gilt:

Satz 4.2. *Für ein Termersetzungssystem $\langle \mathcal{T}, \Rightarrow_R \rangle$ sind folgende Fragen im allgemeinen nicht entscheidbar:*

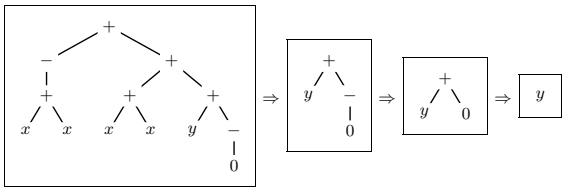


Abb. 4.2. Eine Termersetzungssequenz in Baumdarstellung

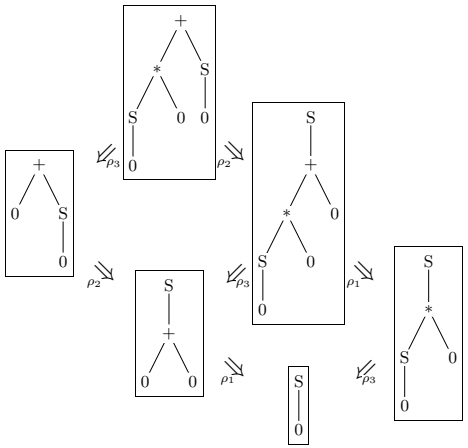


Abb. 4.3. Ersetzungssequenzen auf Term-Bäumen

1. Ist $\Rightarrow_R$ terminierend?
2. Ist $\Rightarrow_R$ konfluent?

Man muss diese Eigenschaften also für jedes System beweisen, oder entscheidbare hinreichende Kriterien dafür finden, wann ein Termersetzungssystem terminierend oder konfluent ist.

Beispiel 4.4 (Termination und Konfluenz). Inspizieren wir das Termersetzungssystem aus Beispiel 4.2.

Man sieht leicht, dass die Normalformen dieses Systems die "Zahlenterme" von der Form "$\mathrm{S}^n\ 0$" sind (für $n \geqslant 0$).

Außerdem ist es so, dass alle Regeln die Symbole "$*$" bzw. "$+$" im Term nach unten bewegen oder eliminieren. Da Terme endlich sind, muss also schließlich die eliminierende Regel anwendbar sein.

Anhand der Ersetzungssequenzen in Abbildung 4.3 kann man sich überlegen, weshalb das System konfluent ist. Alle Regeln enthalten auf der linken Seite genau eines der Symbole "$*$" bzw. "$+$", und in einem Term ist an jedem Symbol "$*$" oder "$+$" nur höchstens eine Regel anwendbar. Die Anwendung einer Regel an einer Stelle in einen Term zerstört keine der anderen Stellen, an denen Regeln angewendet werden könnten. Also können auseinander führende Ersetzungsschritte immer durch Anwenden der jeweils anderen Regel an der anderen Stelle wieder zusammengeführt werden. In der Terminologie der abstrakten Reduktionssysteme ist die Ersetzung "*schwach konfluent*" und – wegen der Termination – auch konfluent.

4.1.4 Systematischer Nachweis von Termination

Die Termination einer Regelmenge kann systematisch untersucht werden, indem versucht wird, eine wohlfundierte partielle Ordnung "$<$" auf Termen (mit Variablen) zu definieren, die mit einem Termersetzungssystem $\langle \mathcal{T}(X), \Rightarrow_R \rangle$ verträglich ist. Dabei gilt eine partielle Ordnung $<$ als *wohlfundiert*, wenn das Ersetzungssystem $\langle \mathcal{T}(X), > \rangle$ terminierend ist.

Definition 4.7 (Reduktionsordnung). Eine wohlfundierte Ordnung $<$ auf Termen ist eine *Reduktionsordnung*, wenn gilt:

1. $<$ ist *gegenüber Substitutionen abgeschlassen*: Aus $t < u$ folgt für beliebige Substitutionen $\sigma\colon X \to \mathcal{T}(X)$, dass $t^\sigma < u^\sigma$.
2. $<$ ist *gegenüber Kontexten abgeschlassen*: Aus $t < u$ folgt für beliebige Kontexte C dass $C[t] < C[u]$.[1]

Eine Reduktionsordnung $<$ ist *verträglich* mit dem Termersetzungssystem $\langle \mathcal{T}(X), \Rightarrow_R \rangle$, wenn für jede Regel $l \to r \in R$ gilt dass $l > r$.

Satz 4.3. *Ein Termersetzungssystem $\langle \mathcal{T}(X), \Rightarrow_R \rangle$ ist terminierend genau dann wenn es eine verträgliche Reduktionsordnung $<$ auf $\mathcal{T}(X)$ gibt.*

Im Folgenden skizzieren wir eine "semantische" Methode – polynomielle Interpretation – und eine "syntaktische Methode" – rekursive Pfadordnungen – zum Konstruieren von verträglichen Reduktionsordnungen .

[1] Ein *Kontext* ist ein Term in dem eine Variable $\triangle$ ("ein Loch") genau einmal auftritt. Dann wird das Einsetzen eines Termes t in C definiert als $C[t] = C^{\sigma_\triangle}$ mit $\sigma_\triangle(\triangle) = t$.

Polynomielle Interpretation

Dabei werden den Funktionssymbolen Interpretationen als Funktionen einer Algebra zugeordnet.

Eine Σ-Algebra $\langle A, \Sigma_a\rangle$ besteht aus einer nicht leeren *Trägermenge* A und aus k-stelligen Funktionen $f_A\colon A^k \to A$ für alle $f \in \Sigma_k$ und alle $k \geqslant 0$.[2] Wir nennen f_A die *Interpretation* von f.

Definition 4.8 (Wohlfundierte monotone Algebra). Eine *wohlfundierte monotone Σ-Algebra* w ist eine Algebra $\langle A, \Sigma_a\rangle$ mit einer wohlfundierten Ordnungsrelation $<$ auf A so dass alle Funktionen der Algebra *strikt monoton* sind. Das heißt, für alle $f \in \Sigma_k$ und alle $a_1, \ldots, a_k, b_1, \ldots, b_K \in A$ mit $a_i < b_i$ für ein i und $a_j = b_j$ für alle $j \neq i$, (wobei $1 \leqslant i \leqslant k$), gilt

$$f_A(a_1, \ldots, a_k) < f_A(b_1, \ldots, b_K)$$

Eine Abbildung $\alpha\colon X \to A$ wird (*Wert-*) *Zuweisung* genannt; sie kann auf eine (*Term-*) *Auswertung* $[\![\cdot]\!]^\alpha\colon \mathcal{T}(X) \to A$ fortgesetzt werden durch die induktive Definition

$$\begin{aligned} [\![x]\!]^\alpha &= \alpha(x) && \text{für } x \in X \\ [\![f(t_1, \ldots, t_k)]\!]^\alpha &= f_A([\![t_1]\!]^\alpha, \ldots, [\![t_k]\!]^\alpha) && \text{für } f \in \Sigma_k, t_1, \ldots, t_k \in \mathcal{T}(X) \end{aligned}$$

Aus dieser Funktion kann man die folgende strikte partielle Ordnung ableiten:[3]

$$t <_A u \iff [\![t]\!]^\alpha < [\![u]\!]^\alpha \text{ für alle Zuweisungen } \alpha\colon X \to A$$

Eine wohlfundierte monotone Algebra $\langle A, \Sigma_a, <\rangle$ ist mit einem Termersetzungssystem $\langle \mathcal{T}(X), \Rightarrow_R\rangle$ *verträglich* wenn für alle Regeln $l \to r \in R$ gilt dass $l >_A r$.

Satz 4.4. *Ein Termersetzungssystem $\langle \mathcal{T}(X), \Rightarrow_R\rangle$ ist terminierend genau dann wenn es eine verträgliche monotone Algebra gibt.*

in den einfacheren Anwendungen dieses Ergebnisses wird die wohl-fundierte monotone Algebra $\langle A, \Sigma_a, <\rangle$ so gewählt, dass die Trägermenge A die natürlichen Zahlen sind, die Funktionen monotone Polynome, und die Ordnung die gewöhnliche Ordnung auf natürlichen Zahlen ist.

Beispiel 4.5 (Termination mit polynomieller Interpretation). Wir betrachten die Regelmenge aus Beispiel 4.2

$$\begin{aligned} \rho_1 &: x + 0 \to x & \rho_2 &: x + \mathrm{S}(y) \to \mathrm{S}(x + y) \\ \rho_3 &: x * 0 \to 0 & \rho_4 &: x * \mathrm{S}(y) \to (x * y) + x \end{aligned}$$

[2] A^k ist die Menge aller k-Tupel $\{(a_1, \ldots, a_k) \mid a_i \in A, 1 \leqslant i \leqslant k\}$ über A.

[3] Eine Ordnung heißt *strikt*, wenn sie *anti-reflexiv* ist, d.h., wenn für alle $a \in A$ gilt, dass $a \not< a$.

Als Trägermenge wählen wir $A = \{n \in \mathbb{N} \mid n < 1\}$. Die Funktionssymbole werden durch folgende Funktionen interpretiert:

$$\begin{aligned} 0_A &= 2 \\ \mathrm{S}_A(x) &= x + 3 && \text{für alle } x \in \mathbb{N} \\ x +_A y &= x + 2y && \text{für alle } x, y \in \mathbb{N} \\ x *_A y &= xy && \text{für alle } x, y \in \mathbb{N} \end{aligned}$$

Diese Funktionen sind monoton. Dazu ist wichtig, den Konstanten Werte größer 1 zu geben, weil sonst die Funktionen $*_A$ und$+_A$ nicht immer monoton wären. Wir bestimmen die Ausdrücke für die linken und rechten Seiten der Regeln:

$$\begin{aligned} [\![l_1]\!] &= x +_A 0_A = x + 2 \cdot 2 = x + 4 \\ [\![r_1]\!] &= x \\ [\![l_2]\!] &= x +_A \mathrm{S}_A(y) = x + 2(y + 3) = x + 2y + 6 \\ [\![r_2]\!] &= \mathrm{S}_A(x +_A y) = (x + 2y) + 3 = x + 2y + 3 \\ [\![l_3]\!] &= x *_A 0_A = x \cdot 2 = 2x \\ [\![r_3]\!] &= 0_A = 2 \\ [\![l_4]\!] &= x *_A \mathrm{S}_A(y) = x(y + 3) = xy + 3x \\ [\![r_4]\!] &= x * Ay +_A x = xy + 2x \end{aligned}$$

Die Funktionen sind verträglich mit den Regeln: Da alle x, y immer größer 1 sind, gilt für beliebige Wertzuweisungen α, dass $[\![l_i]\!]^\alpha > [\![r_i]\!]^\alpha$.

Wenn wir r_4 in die an sich äquivalente Form $x + (x * y)$ bringen, gelingt es interessanterweise nicht, dafür eine polynomiale Interpretation zu finden.

Rekursive Pfadordnungen

Definition 4.9. Eine strikte Odnung auf einer Menge von Funktionssymbolen Σ wird Vorrangrelation genannt (engl.: *precedence*).

Wenn wir Vorrangrelationen auf Terme fortsetzen, müssen wir bei mehr als einstelligen Funktionen $f \in \Sigma_k$ eine Termordnung auf die Argumentfolgen von Termen $f(t_1, \ldots, t_k)$ und $f(u_1, \ldots, u_k)$ fortsetzen. Dafür kann man entweder eine Multimengen-Ordnung oder eine lexikografische Ordnung nehmen. Damit dies für jedes Funktionssymbol individuell bestimmt werden kann, definieren wir den Status.

Definition 4.10 (Status). Die Statusfunktion τ bildet jedes Funktionssymbol $f \in \Sigma_k$ entweder ab auf "mul" oder auf "lex_π", wobei π eine Permutation der Menge $\{1, \ldots, k\}$ ist.

Für jede partielle Termordnung $<$ definiert $<^{\tau(f)}$ eine Ordnung auf Termtupeln der Länge $k > 1$. Wenn $\tau(f) = \mathrm{mul}$, ist dies die Multimengen-Fortsetzung[4]

[4] Multimengen sind ungeordnete Ansammlungen $[s_1, \ldots, s_k]$ von Elementen einer Basismenge S, in denen ein Element mehrmals auftreten kann. Aus $i \neq j$ folgt also

$$\langle t_1, \ldots, t_k\rangle <^{\text{mul}} \langle u_1, \ldots, u_k\rangle \text{ gdw. } [t_1, \ldots, t_k] <_{\#} [u_1, \ldots, u_k]$$

Wenn $\tau(f) = \text{lex}_\pi$, ist dies die lexikographische Fortsetzung auf k-Tupel: [5]

$$\langle t_1, \ldots, t_k\rangle <^{\text{lex}_\pi} \langle u_1, \ldots, u_k\rangle \text{ gdw. } \langle t_{\pi(1)}, \ldots, t_{\pi(k)}\rangle <_{\mathcal{L}} (u_{\pi(1)}, \ldots, u_{\pi(k)})$$

Satz 4.5. *Für jede Signatur Σ mit einer wohlfundierten Vorrangrelation $\prec$ und einem Status π gibt es genau eine Reduktionsordnug $\prec_{\text{rpo}}$ auf den Termen $\mathcal{T}(X)$ mit folgender Eigenschaft:*

$$\begin{aligned} t \succ_{\text{rpo}} u \text{ g.d.w. } & t = f(t_1, \ldots t_k) \text{ und} \\ & (i)\ \ t_i = u \text{ oder } t_i \succ_{\text{rpo}} u \text{ für irgend ein } 1 \leqslant i \leqslant k \text{ oder} \\ & (ii)\ u = g(u_1, \ldots, u_n), t \succ_{\text{rpo}} u_i \text{ für alle } 1 \leqslant i \leqslant m \text{ und} \\ & \qquad \text{entweder } (a) f \succ g, \\ & \qquad \text{oder } (b) f = g \text{ und } \langle t_1, \ldots, t_k\rangle \succ^{\tau(f)}_{\text{rpo}} \langle u_1, \ldots, u_n\rangle \end{aligned}$$

Definition 4.11 (Rekursive Pfad-Ordnung). Die Reduktionsordung nach Theorem 4.5 heißt *rekursive Pfad-Ordnung*. Ist der Status aller Funktionssymbole lexikographisch, spricht man auch von einer *lexikographischen Pfad-Ordnung*.

Beispiel 4.6 (Termination mit rekursiver Pfad-Ordnung). Wir betrachten die Regelmenge aus Beispiel 4.2

$$\rho_1 : x + 0 \to x \qquad \rho_2 : x + \mathrm{S}(y) \to \mathrm{S}(x + y)$$
$$\rho_3 : x * 0 \to 0 \qquad \rho_4 : x * \mathrm{S}(y) \to x + (x * y)$$

Wir wählen die Vorrangrelation $\mathrm{S} \prec + \prec *$ und zeigen, dass alle Regeln $l \to r$ dann $l \succ_{\text{rpo}} r$ erfüllen.

$$\begin{aligned} x + 0 &\succ_{\text{rpo}} x && \text{nach Fall (1)} \\ x + \mathrm{S}(y) &\succ_{\text{rpo}} \mathrm{S}(x + y) && \text{g.d.w. (wegen Fall (2)(a))} \\ x + \mathrm{S}(y) &\succ_{\text{rpo}} x \text{ und } x + \mathrm{S}(y) \succ_{\text{rpo}} y && \text{dies gilt nach Fall (1)} \\ x * 0 &\succ_{\text{rpo}} 0 && \text{nach Fall (1)} \\ x * \mathrm{S}(y) &\succ_{\text{rpo}} x + (x * y) && \text{g.d.w. (wegen Fall (2)(a))} \\ x * \mathrm{S}(y) &\succ_{\text{rpo}} x \text{ und } x * \mathrm{S}(y) \succ_{\text{rpo}} y && \text{dies gilt nach Fall (1)} \end{aligned}$$

nicht immer $s_i \neq s_j$. Formal ist eine (endliche) *Multimenge* über einer Basismenge S eine Abbildung $M\colon S \to \mathbb{N}$, wobei $M(s) = 0$ für alle $s \in S$ bis auf endlich viele. *Enthaltensein* in Multimengen wird definiert als $s \in_{\#} M$ genau dann wenn $M(s) < 0$. Die *disjunkte Vereinigung* $M \uplus_{\#} M'$ von Multimengen M, M' über einer Menge S ist definiert als $(M \uplus_{\#} M')(s) = M(s) + M'(s)$ für alle $s \in S$. Für eine Ordnung $<$ auf eine Menge S wird die Fortsetzung auf Multimengen über S definiert als die kleinste transitive Relation $<_{\#}$ für die gilt, dass $M \uplus_{\#} M <_{\#} M \uplus_{\#} [s]$ wenn $\forall x \in_{\#} M' : x < s$. Aus einer Multimenge M enthält man also eine kleinere, wenn man ein Element $s \in_{\#} M$ entfernt und durch beliebig viele kleinere Elemente $x \in M'$ ersetzt.

[5] Für eine strikte Ordnung $<$ auf eine Menge S ist die *lexikographische Fortsetzung* auf k-Tupel (mit $k > 1$) definiert als $\langle s_1, \ldots, s_k\rangle <_{\mathcal{L}} \langle s'_1, \ldots, s'_k\rangle$ gdw. für ein $i < k$ gilt, dass $\langle s_1, \ldots, s_i\rangle = \langle s'_1, \ldots, s'_i\rangle$ und $s_{i+1} < s'_{i+1}$. Wann immer $<$ eine strikte Ordnung ist, gilt das auch für $<_{\mathcal{L}}$.

Bemerkenswerterweise gilt dies auch, wenn die Operanden auf der rechten Seite der letzten Regel vertauscht werden. Für diese Regel war es ja nicht möglich gewesen, Termination mittels polynomieller Interpretation zu zeigen.

BAUSTELLE Siehe [Zan03].

4.1.5 Nicht überlappende Regelmengen

Beispiel 4.7 (Überlappende Regeln, [KdV03]). Gegeben sei folgende Regelmenge:

$$\rho_5 : f(g(x), y) \to x \qquad \rho_6 : g(a) \to b$$

Die linken Seiten dieser Regeln können in einem Term *überlappen*, wie in Abbildung 4.4 gezeigt. Dabei zerstört die Anwendung einer Regel jeweils die Möglichkeit, auf den resultierenden Term die andere anzuwenden, und die Ersetzung ist nicht konfluent wie hier. Der Baum des Terms $f(g(a), y)$ oben in der Mitte ist der *Kern einer kritischen Überlappung*, weil er nur aus Bestandteilen der linken Regelseiten besteht. Dir Terme $F(b, y)$ und a, deren Bäume unten links und rechts abgebildet sind, heißen dann ein *kritisches Paar*.

Eine Regel kann auch mit sich selbst überlappen, wie z.B. die Regel $g(g(x)) \to x$. Dann ist eine Überlappung nur kritisch, wenn einer der Terme ein echter Unterterm des anderen ist, wie in $g(g(g(x)))$. (Das kritische Paar besteht in diesem Fall aus den Termen $g(x)$ und $g(x)$ und ist eigentlich gar nicht kritisch.)

Das Überlappen ist dann *kritisch*, wenn nicht nur eine Variable der einen Regel mit der Wurzel der anderen Regel überlappt, sondern auch ein Funktionssymbol (wie g in diesem Fall). Dann nennen wir $\langle f(b, y), a\rangle$ ein kritisches Paar.

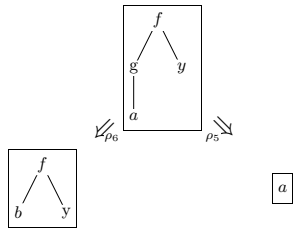


Abb. 4.4. Überlappende Regeln

Satz 4.6. *Ein Termersetzungssystem $\langle \mathcal{T}, \Rightarrow_R\rangle$ ist schwach konfluent, wenn alle Regeln in R einander nicht überlappen.*

Überlappungsfreiheit reicht nicht aus, um auch Konfluenz zu zeigen. Siehe dazu Übung 2.7.20 in [KdV03]. Dazu müssen die Termersetzungsregeln *linkslinear* sein, d.h., jede Variable darf auf der linken Regelseite höchstens einmal auftreten.

Definition 4.12. Eine Menge R von Termersetzungsregel heißt *orthogonal*, wenn alle ihre Regeln linkslinear sind und einander nicht überlappen.

Satz 4.7. *Ein Termersetzungssystem $\langle \mathcal{T}, \Rightarrow_R \rangle$ mit orthogonaler Regelmenge R ist konfluent.*

4.1.6 Kritische Paare

Auch wenn eine Regelmenge überlappende Regeln enthält, kann sie trotzdem schwach konfluent sein, nämlich dann, wenn alle *kritischen Paare* von allen Regeln durch Sequenzen von Termersetzungsschritten wieder zusammengeführt werden können. In Abbildung 4.4 ist der mittlere Termbaum der Kern einer kritischen Überlappung, und die Terme links und rechts bilden ein kritisches Paar. Nach dem *Critical Pair Lemma* von Knuth-Bendix müsste aus dem Paar der gleiche Term abgeleitet werden können, damit die Termersetzung trotzdem schwach konfluent ist. In diesem Beispiel ist das nicht der Fall. In so einer Situation könnte man die Regelmenge ergänzen, so dass dies gilt, zum Beispiel durch Hinzunehmen der Regel

$$f(b, y) \to a$$

Hierbei muss man einerseits überlegen, ob diese Regel sinnvoll ist, und außerdem überprüfen, ob sie mit anderen Regeln überlappt. Diese Prozedur wird nach ihren Erfindern *Knuth-Bendix Completion* genannt. Auch mit diesem Stoff kann man ein ganzes Kapitel füllen.

BAUSTELLE! Siehe [Bet03]

4.2 Von Regeln zu funktionalen Programmen

Termersetzung wird möglichst einfach definiert, um Berechnungen ohne unnützen Ballast zu studieren. Zwar könnte man damit auch "praktisch" programmieren, man würde aber vermutlich einige Konzepte vermissen, die man aus "richtigen" Programmiersprachen kennt.

4.2.1 Typen

Bisher sind Funktionssymbole nur grob nach der Anzahl ihrer Parameter klassifiziert, aber nicht nach den Typen der Parameter und des Resultats.

Beispiel 4.8. Die Funktionssymbole aus Beispiel 4.2 sollen Funktionen auf natürlichen Zahlen definieren. Ihre Typen könnte man also so spezifizieren:

$$0 :: \mathrm{Nat}, \quad S :: \mathrm{Nat} \to \mathrm{Nat}, \quad (+), (*) :: \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Nat}$$

Im einfachsten Fall betrachtet man nur eine Menge S von Typnamen oder *Sorten*, und klassifiziert Funktions-Parameter, -Resultate und auch Variablen und Terme nach diesen Sorten. Wenn $s \in S$, schreiben wir $t :: s$, um auszudrücken, dass der Term t von der Sorte s ist. Die Bildung von Termen muss dann so modifiziert werden:

$$\frac{x :: s \in X}{x :: s \in \mathcal{T}(X)} \qquad \frac{f :: s_1 \cdots s_k \to s \in \Sigma,\ t_i :: s_i, \in \mathcal{T}(X),\ 1 \leqslant i \leqslant k, k \geqslant 0}{(f\ t_1 \cdots t_k) :: s \in \mathcal{T}(X)}$$

Die Regeln müssen aus Termen der gleichen Sorte gebildet werden, und auch in Termersetzungsschritten $t \Rightarrow_R u$ sind t und u von der gleichen Sorte. Diese Art von Termersetzung heißt *mehrsortig* (*many-sorted*). Manchmal werden die Sorten auch mit einer *Untersortenrelation* ausgestattet, wie in [GM92], um gewisse Eigenschaften objektorientierter Typisierungen zu modellieren.

Will man auch Funktionen höherer Ordnung modellieren, d.h., Funktionen, deren Parameter oder Resultate selber Funktionen sind, wird es etwas komplizierter. Man muss funktionale Typen modellieren, und bildet *Typausdrücke* $\mathcal{S}$ über einer Menge $\bar{S}$ von Basissorten wie folgt:

$$\frac{\bar{s} \in \bar{S}}{\bar{s} \in \mathcal{S}} \qquad \frac{s_1, s_2 \in \mathcal{S}}{(s_1 \to s_2) \in \mathcal{S}}$$

Der Typausdruck $s_1 \to s_2$ bezeichnet den *Funktionsraum* der Funktionen von s_1 nach s_2. Um Klammern zu sparen, nehmen wir an, $s_1 \to s_2 \to s_3$ stünde für $(s_1 \to (s_2 \to s_3))$ und geben Funktionen mit mehreren Parametern den *ge-curry-ten*[6] Typ

$$s_1 \to \cdots \to s_k \to s$$

In den meisten funktionalen Sprachen können Typausdrücke auch Variablen enthalten, für die beliebige Typen eingesetzt werden können, und die Basistypen enthalten nicht nur Namen von einfachen "konstanten" Typen wie "nat", sondern auch Namen für parametrisierte Typen wie "list s", der Typ der Listen mit Elementen aus s. Die Basistypen müssen also nach ihrer Stelligkeit klassifiziert werden: "nat" hat die Stelligkeit 0 und "list" die Stelligkeit 1. Das kennen wir schon von den Funktionssymbolen (wie sie ursprünglich definiert waren). Wir setzen also voraus, dass die Basissorten $\bar{S}$ in paarweise disjunkte Mengen $\bar{S}_0$ von *Typkonstanten* und k-stelligen *Typkonstruktoren* $\bar{S}_k$ (für $k > 0$) aufgeteilt ist. Die Regeln zum Bilden von Typausdrücken mit Variablen sehen so aus:

$$\frac{\alpha \in Y}{\alpha \in \mathcal{S}(Y)} \qquad \frac{\bar{s} \in \bar{S}_k, s_1, \ldots, s_k \in \mathcal{S}(Y)}{(\bar{s}\ s_1 \cdots\ s_k) \in \mathcal{S}(Y)} \qquad \frac{s_1, s_2 \in \mathcal{S}(Y)}{(s_1 \to s_2) \in \mathcal{S}(Y)}$$

[6] Nach dem Logiker Haskell Curry.

Parametrisierte Typausdrücke erlauben es, *polymorphe Funktionen* und *Funktionen höherer Ordnung* zu vereinbaren wie die Funktionen

$$\text{Cons} :: \alpha \to \text{list}\,\alpha \to \text{list}\,\alpha \qquad \text{map} :: (\alpha \to \beta) \to \text{list}\,\alpha \to \text{list}\,\beta$$

Cons ist ein Wertkonstruktor, der ein Element vor eine Liste hängt, für Listen beliebigen Elementtyps α. Die Funktion map hat eine Funktion f als Parameter und ersetzt in jeder Liste die Elemente a durch $f(a)$.

Terme bilden wir auf *applikative Weise*, indem wir alle Funktionssymbole als Konstanten (mit Stelligkeit 0 und Typ aus $\mathcal{S}$) ansehen, und nur eine zweistellige Operation

$$(\cdot) :: (\alpha \to \beta) \to \alpha \to \beta$$

namens *Funktionsanwendung* vorsehen. Die Funktionsanwendung wird als links geklammert angesehen, d.h., $f \cdot t_1 \cdot t_2$ stünde für $((f \cdot t_1) \cdot s_2)$. Meistens lassen wir das Zeichen für die Funktionsanwendung aber ganz weg und schreiben nur $f\,t_1\,t_2$.

Beispiel 4.9. Die Funktionssymbole aus Beispiel 4.2 könnte man mit folgenden Typausdrücken spezifizieren:

$$0 :: \text{Nat}, \quad S :: \text{Nat} \to \text{Nat}, \quad (+), (*) :: \text{Nat} \to \text{Nat} \to \text{Nat}$$

Die Regeln würden applikativ so geschrieben:

$$\rho_1 : (+) \cdot x \cdot 0 \to x \qquad \rho_2 : (+) \cdot x \cdot (\text{S} \cdot y) \to \text{S} \cdot ((+) \cdot x \cdot y)$$
$$\rho_3 : (*) \cdot x \cdot 0 \to 0 \qquad \rho_4 : (*) \cdot x \cdot (\text{S} \cdot y) \to (+) \cdot x \cdot ((*) \cdot x \cdot y)$$

Dies ist etwas umständlich. Es ist üblich, den Anwendungsoperator wegzulassen, also einfach die Parameter hinter die Funktion zu schreiben. Außerdem spendieren wir uns wieder die Infixschreibweise für Operatoren:

$$\rho_1 : x + 0 \to x \qquad \rho_2 : x + (\text{S}\,y) \to \text{S}(x + y)$$
$$\rho_3 : x * 0 \to 0 \qquad \rho_4 : x * (\text{S}\,y) \to x + (x * y)$$

Damit sieht wieder alles fast so aus wie vorher. Allerdings ist jetzt auch "S" und "$(x\,+)$" Terme (beide vom Typ Nat $\to$ Nat). Funktionen können also auch unangewendet oder – wie "+" – nur partiell angewendet in einem Term stehen.

4.2.2 Konstruktorfunktionen und definierte Funktionen

In Beispiel 4.2 gibt es zwei verschiedene Arten von Funktionssymbolen:

- Die Symbole "0" und "S" repräsentieren "Daten", weil die Normalformen des Systems aus diesen Symbolen bestehen. Hier sind sogar alle Terme, die nur aus diesen beiden Symbolen bestehen, Normalformen.

- Die Symbole "+" und "$*$" dagegen repräsentieren zu berechnende Funktionen, denn es gibt Ersetzungsregeln, die alle Auftreten dieser Symbole eliminieren. Es gibt keine Normalform, die eines dieser Symbole enthält.

In funktionalen Sprachen werden diese beiden Arten von Funktionssymbolen strikt getrennt:

- Daten repräsentierende Funktionssymbole sind Wert-konstruierende Funktionen, auch kurz *Wertkonstruktoren* genannt. Das Symbol "0" konstruiert den Term für die Zahl 0, und die Funktion "S" konstruiert für einen Term, der eine Zahl n repräsentiert, den Term für die Zahl $n + 1$. Namen von Wertkonstruktoren werden in vielen funktionalen Sprachen ausgezeichnet, indem sie groß geschrieben werden.
- Zu berechnende Funktionssymbole werden *definierte Funktionen* genannt. Die Regeln des Ersetzungssystems berechnen ihre Werte, wie bei "+" und "$*$". Namen von definierten Funktionen sind entweder Operatorsymbole oder klein geschriebene Bezeichner.

Aus praktischen Erwägungen werden sowohl Konstruktoren als auch definierte Funktionen eingeschränkt.

Definition 4.13. Ein Wertkonstruktor ist *frei*, wenn verschiedene Terme über den Konstruktorsymbolen immer auch verschiedene Werte repräsentieren.

In funktionalen Sprachen wird verlangt, dass alle Wertkonstruktoren frei sind. Für die Konstruktoren aus Beispiel 4.2 ist dies der Fall. Im Allgemeinen ist dies aber eine erhebliche Einschränkung.

Beispiel 4.10. Wir können Mengen von natürlichen Zahlen mit den Wertkonstruktoren "Empty :: natset" (für die leere Menge) und "Insert :: Nat $\to$ natset $\to$ natset" spezifizieren. Ganz ohne Gleichungen definiert dies aber eher Listen statt Mengen, weil gilt:

$$\text{Insert 1 (Insert 2 Empty)} \neq \text{Insert 2 (Insert 1 Empty)}$$
$$\text{Insert 1 (Insert 1 Empty)} \neq \text{Insert 1 Empty}$$

Um zu respektieren, dass Mengen ungeordnet und idempotent sind, müsste man Regeln auf den Wertkonstruktoren definieren:

$$\text{Insert}\,x\,(\text{Insert}\,y\,\text{Empty}) \to \text{Insert}\,y\,(\text{Insert}\,x\,\text{Empty})$$
$$\text{Insert}\,x\,(\text{Insert}\,x\,\text{Empty}) \to \text{Insert}\,x\,\text{Empty}$$

Dann wären die Konstruktoren Insert und Empty nicht *frei*; typischerweise können Datentypen mit "unfreien" Konstruktoren in funktionalen Sprachen nicht so definiert werden.[7]

[7] Sie können aber als *abstrakte Datentypen* definiert werden. Dann werden die freien Konstruktoren verborgen und nur abstrakte Konstruktoren exportiert. Meist darf

Des Weiteren wird von den definierten Funktionssymbolen verlangt, dass sie durch Regeln "hinreichend vollständig" ("*sufficiently complete*") beschrieben sind. Darunter ist zu verstehen, dass die Regeln diese Symbole vollständig eliminieren. Die definierten Funktionssymbole aus Beispiel 4.2 sind hinreichend vollständig. Ist dies nicht der Fall, wird die Funktion als partiell engesehen, und implizit eine Fehlerregel für die nicht definierten Fälle hinzugefügt.

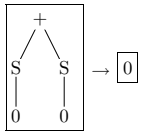
Darüber hinaus werden die Regeln noch weiter eingeschränkt:

- Sie müssen *Konstruktor-basiert* sein, d.h., außer einem definierten Symbol als Wurzel dürfen die linke Seiten der Regeln keine weiteren definierten Symbole enthalten.
- Sie müssen *links-linear* sein, d.h., Variablen dürfen auf den linken Seiten der Regeln jeweils höchstens einmal auftreten.

Beispiel 4.2 erfüllt auch diese Bedingungen.

Freiheit der Konstruktoren und Konstruktor-Basiertheit schließen schon viele Fälle von kritischen Überlappungen aus, aber doch nicht alle: Die unsinnige Regel

$$\rho_7 : S\,0 + S\,0 \to 0$$



$\to \boxed{0}$

ist auch Konstruktor-basiert, überlappt aber kritisch mit Regel ρ_2 von Beispiel 4.2; zusammen wären diese Regeln nicht mehr konfluent.

Man könnte von Konstruktor-basierten Regeln auch noch Freiheit von kritischen Überlappungen fordern, aber praktisch wird das meistens anders gelöst. Wie, steht im nächsten Unterabschnitt.

4.2.3 Strategien

Auch mit eingeschränkten Regeln bleibt die Termersetzungsrelation "$\Rightarrow_R$" im Allgemeinen nichtdeterministisch, ein Term t kann also auf verschieden Weise ersetzt werden, z.B. $t \Rightarrow_\rho u$ und $t \Rightarrow_{\bar\rho} \bar u$ gibt es zwei Freiheitsgrade:

- Die anzuwendende Regel kann ausgesucht werden, und
- die Stelle, an der sie angewendet wird, kann gewählt werden.

Wenn Termersetzung terminierend und konfluent ist, hat das auf das Ergebnis keinen Einfluss, weil jeder Term genau eine Normalform hat. Allerdings können wir Termination nicht immer voraussetzen, und dann könnten einige

man abstrakte Konstruktoren jedoch nicht in Mustern (auf der linken Seite von Definitionen) verwenden.

Ersetzungssequenzen nie terminieren. Andernfalls beträfe dies die Effizienz, weil die normalisierenden Sequenzen unterschiedlich lang sein könnten.

Dies ist die Motivation für *Strategien*. Allgemein ist eine Strategie eine Vorschrift, die die Wahl der nächsten anzuwendenden Regel und/oder der nächsten Anwendungsstelle für eine Regel einschränkt, quasi eine *Metaregel*.

Beispiele für Strategien sind:

- Ordne die Regeln total, beispielsweise nach der Reihenfolge, in der sie angegeben wurden, und wähle immer die erste passende Regel aus.
- Wähle immer die Anwendungsstelle "links außen" (*outermost*).
- Wähle immer die Anwendungsstelle "links innen" (*innermost*).

Für die Regeln aus Beispiel 4.2 spielt die Reihenfolge keine Rolle, weil sie nicht überlappen, und an jeder Anwendungsstelle höchstens eine von ihnen anwendbar ist. In Abbildung 4.3 folgt die Sequenz mit der Schrittfolge $\langle \rho_2, \rho_1, \rho_3 \rangle$ der oben-links-Strategie, und die Sequenz mit der Schrittfolge $\langle \rho_3, \rho_2, \rho_1 \rangle$ der unten-links-Strategie (und die mit der Schrittfolge $\langle \rho_2, \rho_3, \rho_1 \rangle$ keiner von beiden).

Fakt 4.8 *Seien $\overset{\text{i}}{\Rightarrow}_R$ und $\overset{\text{o}}{\Rightarrow}_R$ die Strategien unten-links bzw. oben-links, jeweils mit geordneten Regeln.*

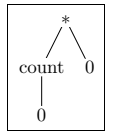
1. *Sowohl $\overset{\text{i}}{\Rightarrow}_R$ als auch $\overset{\text{o}}{\Rightarrow}_R$ sind deterministisch.*
2. *Die Strategie $\overset{\text{o}}{\Rightarrow}_R$ ist normalisierend.*
3. *Die Strategie $\overset{\text{i}}{\Rightarrow}_R$ ist nicht normalisierend.*

Ein Beispiel erläutert, weshalb die Strategie $\overset{\text{i}}{\Rightarrow}_R$ nicht normalisierend ist.

Beispiel 4.11. Sei das Funktionssymbol count :: Nat → Nat definiert mit

$$\text{count}\ x = \text{count}(\text{S}\ x)$$

Betrachten wir den Baum



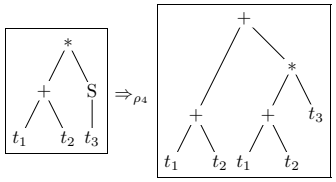
Dann berechnet $\overset{\text{o}}{\Rightarrow}_R$ in einem Schritt die Normalform 0, während $\overset{\text{i}}{\Rightarrow}_R$ nicht terminiert.

Trotzdem wird bei der Auswertung von Ausdrücken in Programmiersprachen auch in vielen funktionalen Sprachen die unten-links-Strategie angewendet, die auch unter den Bezeichnungen *strikt* und *call-by-value* bekannt ist. In funktionalen Sprachen muss dann jedoch eine Ausnahme gemacht werden. Bei Auswahlfunktionen wie

if c **then** t **else** e

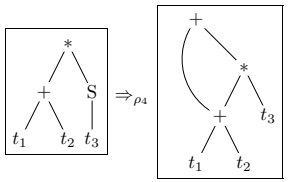
dürfen die Terme t und e nicht nach der Strategie unten-links ausgewertet werden, sondern nur einer von beiden, nachdem die Bedingung c vollständig ausgewertet wurde.

Die oben-links-Strategie ist etwas schwieriger zu implementieren und wertet Terme auch nicht immer sehr effizient aus. Betrachten wir den Term



In diesem Schritt wird der Teilterm "$t_1 + t_2$" kopiert und damit nicht nur "Daten", sondern auch "Arbeit", denn die definierten Funktionssymbole (wie "+") in diesem Term müssen nun zweimal ausgewertet werden. Und dies, obwohl wegen der Konfluenz des Systems klar ist, dass die Auswertung die gleiche Normalform liefern wird.

Statt den Teilterm zu kopieren, könnten wir auch zweimal auf ihn verweisen:



Der Ersetzungsschritt liefert dann statt eines Termbaumes einen *Termgraphen*. Die Arbeit wird nicht kopiert, sondern die Früchte einmaliger Arbeit werden nur an zwei Stellen verwendet. So wird die Termersetzung in funktionalen Sprachen implementiert. Die Strategie heißt "verzögert". Sie ist eine Variation von "oben-links", wobei aber ein Term höchstens einmal ausgewertet wird, und das Ergebnis dann für alle seine Kopien eingesetzt wird. Verzögerte Auswertung heißt auch *call-by-need"* oder "*lazy*".

4.2.4 Termersetzungssprachen

Die meisten Termersetzungssprachen, die für das Seminar vorgeschlagen werden, erweitern Termersetzungssysteme um die in diesem Abschnitt vorgeschlagenen Konzepte – Typen, Konstruktorbasiertheit und Strategien. Manche verwenden keine feste Strategie, sondern erlauben, Strategien in der Sprache selbst zu definieren.

4.3 Klausellogik

In der Klausellogik wird eine Teilmenge der Prädikatenlogik erster Stufe dazu verwendet, Prädikate zu beschreiben. Prädikate sind Funktionen, die Wahrheitswerte liefern; wenn ein Prädikat den Wert *wahr* liefert, sagen wir, *das Prädikat gilt*, andernfalls, es gelte nicht. Die Auswertung einer Menge von Prädikat-Definitionen zählt die Menge von Werten auf, für die ein Prädikat (oder eine Konjunktion von Prädikaten) gilt.

- Termersetzungsregeln beschreiben *Funktionen* – Fakten und Schlussregeln beschreiben *Relationen*.
- Termersetzung berechnet Funktionswerte – Resolution zählt Relationen auf.

4.3.1 Formeln und Klauselmengen

Die Formeln der Prädikatenlogik werden über den aus Unterabschnitt 4.1.1 bekannten Termen $\mathcal{T}(X)$ aufgebaut. (Wir verwenden also ungetypte Terme erster Ordnung, nicht die in Unterabschnitt 4.2.1 eingeführten mehrsortigen oder applikativen Terme.)

Dazu kommt eine Menge Π von *Prädikatsymbolen*, die nach ihrer Stelligkeit $k \geqslant 0$ in Mengen $\Pi_0, \Pi_2, \ldots$ aufgeteilt ist.

Die *prädikatenlogischen Formeln* $\mathcal{F}(X)$ sind die kleinste Menge, die folgende Eigenschaften erfüllt:

$$\frac{p \in \Pi_k, t_1, \ldots, t_k \in \mathcal{T}(X)}{p(t_1, \ldots, t_k) \in \mathcal{F}(X)}(\text{AF}) \qquad \frac{\varphi \in \mathcal{F}(X)}{(\neg\varphi) \in \mathcal{F}(X)}(\text{neg})$$

$$\frac{\varphi, \rho \in \mathcal{F}(X)}{(\varphi \wedge \rho), (\varphi \vee \rho), (\varphi \supset \rho), (\varphi \leftrightarrow \rho) \in \mathcal{F}(X)}(\text{op}) \qquad \frac{x \in X, \varphi \in \mathcal{F}(X)}{(\forall x : \varphi), (\exists x : \varphi) \in \mathcal{F}(X)}(\text{quant})$$

Hierbei benutzen wir die üblichen Präzedenzen der logischen Konnektoren

$$(\neg) \succ (\wedge) \succ (\vee) \succ (\supset) \succ (\leftrightarrow) \succ (\forall) \succ (\exists)$$

um beim Schreiben von Formeln Klammern zu sparen.

Wir definieren die Menge $Fvar(\varphi)$ der *freien Variablen* in einer Formel φ rekursiv wie folgt:

$$\begin{aligned} Fvar(p(t_1, \ldots, t_k)) &= \textstyle\bigcup_{i=1}^{k} Var(t_i) \\ Fvar(\neg\varphi) &= Fvar(\varphi) \\ Fvar(\varphi \oplus \rho) &= Fvar(\varphi) \cup Fvar(\rho) \qquad \text{für } \oplus \in \{\wedge, \vee, \supset, \leftrightarrow\} \\ Fvar(\forall x : \varphi) &= Fvar(\exists x : \varphi) = Fvar(\varphi) \setminus \{x\} \end{aligned}$$

Eine Formel φ heißt *geschlossen* wenn $Fvar(\varphi) = \{\}$ (und *offen* sonst).

Weiteres zu allgemeiner Prädikatenlogik ist in [NM95] nachzulesen. Hier werden wir uns nur mit sehr eingeschränkten prädikatenlogischen Formeln befassen, sogenannten *Klauseln*.

4.3.2 Klauseln und Ziele

Klauseln sind Formeln

$$\forall x_1 : \cdots \forall x_k : \exists y_1 : \cdots \exists y_n : P_1 \wedge \cdots \wedge P_n \supset P_0 \qquad (k \geqslant 0, n \geqslant 0)$$

Hierbei sind die P_i *atomare Formeln* der Bauart "$p_i(t_{i,1}, \ldots, t_{i,k_i})$", und die Formel ist *abgeschlossen*, so dass $Fvar(P_0) = \{x_1, \ldots, x_k\}$ und $Fvar(P_1 \wedge \cdots \wedge P_k) = Fvar(P_0) \cup \{y_1, \ldots, y_n\}$. Weil das (Nicht-) Auftreten der Variablen in P_0 festlegt, ob sie mit $\forall$ oder $\exists$ quantifiziert sind, werden die Quantoren weggelassen, die Seiten der Regel vertauscht, und die Umkehrung des Zeichens "$\supset$" als "$\leftarrow$" geschrieben:

$$P_0 \leftarrow P_1 \wedge \cdots \wedge P_n \qquad (n \geqslant 0)$$

So eine Klausel ist zu lesen als:

> P_0 *gilt (für alle* $x_1, \ldots, x_k$*),*
> *wenn (es* $y_1, \ldots, y_n$ *gibt, so dass)* P_1*, ... und* P_n *gelten*

Ist $n = 0$ und damit die rechte Seite leer, schenken wir uns den Pfeil. Eine solche "bedingungslose" Klausel wird *Faktum* (engl. *fact*) genannt.

Klauseln und Fakten werden benutzt, um *Widerspruchsbeweise* zu führen. Man versucht, eine Frage (engl. *goal*) zu beantworten von der Form

$$Q_1 \wedge \cdots \wedge Q_n ? \qquad (n \geqslant 0)$$

Jede der Q_i heißt *Teilfrage* (engl. *subgoal*).

4.3.3 Unifikation

Um eine Klausel mit linker Seite $P_0 = p(t_1, \ldots, t_k)$ auf ein Prädikat $Q_i = q(u_1, \ldots, u_m)$ in einer Anfrage anzuwenden, müssen Namen und Stelligkeit der Prädikate gleich sein, h.h., $p = q$, $k = m$, und die Terme t_i und u_i für $1 \leqslant i \leqslant k = m$ müssen unifizierbar sein. Das schreiben wir als

$$t_1 \stackrel{\circ}{=} u_1 \wedge \cdots \wedge t_k \stackrel{\circ}{=} u_n$$

Zwei Terme t und u sind unifizierbar, wenn es eine Substitution $\theta \colon X \to \mathcal{T}(X)$ gibt, die beide Terme gleich macht, also $t^\theta = u^\theta$. Dann nennen wir θ einen *Unifizierer* von t und u.

Genauer interessieren wir uns für den *allgemeinsten Unifizierer* zweier Terme (englisch: *most general unifier* kurz "mgu"). Das ist der Unifizierer, der so wenig wie möglich ersetzt und in allen anderen Unifizierern enthalten ist.

Die folgenden Regeln nach [KdV03] bestimmen, ob eine Menge von Termgleichungen unifizierbar ist. Dabei nehmen wir der Einfachheit halber an, dass die Variablenmengen der Prädikate P_0 und Q_i disjunkt sind.

$$\frac{f\, t_1 \cdots t_k \stackrel{\circ}{=} f\, u_1 \cdots u_k \wedge E}{t_1 \stackrel{\circ}{=} u_1 \wedge \cdots \wedge t_k \stackrel{\circ}{=} u_k \wedge E} \quad \text{(match)}$$

$$\frac{f\, t_1 \cdots t_k \stackrel{\circ}{=} g\, u_1 \cdots u_m \wedge E \quad (f \neq g)}{\textbf{failure}} \quad \text{(fail)}$$

$$\frac{x \stackrel{\circ}{=} t \wedge E, x \in Var(t)}{\textbf{failure}} \quad \text{(occur check)}$$

$$\frac{x \stackrel{\circ}{=} t \wedge E, x \notin Var(t)}{x \stackrel{\circ}{=} t \wedge E^{\{x \mapsto t\}}} \quad \text{(substitution)}$$

In der letzten Regel bedeutet $E^{\{x \mapsto t\}}$, dass in allen Termgleichungen von E jedes Auftreten der Variablen x durch den Term t ersetzt wird.

Wenn die Unifikation einer Gleichungsmenge gelingt, liefert sie eine Menge von Lösungen

$$x_1 \stackrel{\circ}{=} t_1 \wedge \cdots \wedge x_n \stackrel{\circ}{=} t_n$$

Dann wird eine Funktion $\theta \colon X \to \mathcal{T}(X)$ mit

$$\theta(x) = \begin{cases} t_i & \text{wenn } x = x_i, 1 \leqslant i \leqslant n \\ x & \text{sonst} \end{cases}$$

allgemeinster Unifizierer des Unifizierungsproblems genannt.

Der allgemeinste Unifizierer wird bestimmt als die mit der letzten Regel aufgebaute Substitution – wenn sie existiert.

Bei der Unifikation müssen die Terme textuell gleich gemacht werden; die Funktionssymbole und Konstantensymbole werden als freie Konstruktoren von Werten aufgefasst.

4.3.4 Resolution

Resolutionsschritte unifizieren ein Prädikat der Anfrage mit der linken Seite einer Klausel und ersetzen dieses Prädikat durch die rechte Klausel-Seite, wobei der allgemeinste Unifizierer auf alle Prädikate der Anfrage und der rechten Klauselseite angewendet wird. Resolution geschieht also nach der folgenden Regel:

$$\frac{Q_1 \wedge \cdots \wedge Q_n, P_0 \leftarrow P_1 \wedge \cdots \wedge P_n \in R, \theta = \mathrm{mgu}(Q_i, P_0)}{Q_1^\theta \wedge \cdots \wedge Q_{i-1}^\theta \wedge P_1^\theta \wedge \cdots \wedge P_n^\theta \wedge Q_{i+1}^\theta \wedge \cdots \wedge Q_n^\theta}$$

Die Resolution ist erfolgreich, wenn alle Teilfragen eliminiert werden können; dann ist die ursprüngliche Frage mit *ja* beantwortet. Sonst schlägt die Resolution fehl. Da jeder Resolutionsschritt eine beliebige Klausel auf eine beliebige Teilfrage anwendet, könnte es jedoch noch eine andere Folge von Resolutionsschritten geben, die erfolgreich ist. Deshalb müssen bei Fehlschlag systematisch die weiteren möglichen Resolutionen ausprobiert werden. Diese Prozedur nennt man *Zurücksetzen* (engl. *backtracking*).

Im Falle eines Erfolgs wird im Allgemeinen nicht nur die Anfrage mit *Ja* beantwortet, sondern auch noch ein *Beispiel* für eine Belegung der Variablen in der Anfrage gegeben, für die die Anfrage mit ja beantwortet werden kann. Die Belegung ist die Kombination der allgemeinsten Unifizierer, die in den Resolutionsschritten benutzt wurden. Durch Zurücksetzen können dann noch weitere Beispiele angefordert werden.

4.3.5 Strategien

Bei der Auswahl des nächsten Resolutionsschritts gibt es zwei Freiheitsgrade:

- die anzuwendende Regel
- die zu ersetzende Teilfrage

Ähnlich wie bei den Strategien der Termersetzung werden die Fakten und Regeln in der Reihenfolge angewendet, in der sie aufgeschrieben wurden.

Für die Auswahl der nächsten zu ersetzenden Teilfrage gibt es zwei Strategien:

- Bei der *Tiefensuche* (engl. *depth-first search*) wird immer die am weitesten links stehende Teilfrage ausgewählt und versucht, eine Regel auf sie anzuwenden.
- Bei der *Breitensuche* (engl. *breadth-first search*) werden auf alle Teilfragen der eingegebenen Frage (von links nach rechts) Regeln angewendet, bevor für die daraus resultierende "Anfrage der zweiten Generation" genauso verfahren wird.

Beide Strategien sind deterministisch, auch wenn beide mehr als einmal erfolgreich sein können. Darüber hinaus gilt:

Satz 4.9. *Für die Resolutionsstrategien gilt:*

1. *Breitensuche ist normalisierend, d.h., jede erfolgreiche Resolutionssequenz wird irgendwann gefunden.*
2. *Tiefensuche ist nicht normalisierend.*

Trotzdem wird eher die Tiefensuche implementiert, weil Breitensuche für praktische Anwendungen zu ineffizient ist.

Beispiel 4.12 (Die Windsors). Genealogische Daten sind im Wesentlichen Relationen und eignen sich deshalb hervorragend als Beispiel.

Ein Auszug aus der Abstammung der Familie Windsor kann durch die untenstehenden Klauseln in der Schreibweise von PROLOG ansatzweise beschrieben werden.

```
mother(queenMum,elizabethII).
mother(elizabethII,charles).
mother(elizabethII,edward).
```



```
mother(elizabethII,andrew).
mother(elizabethII,anne).
father(charles,william).
father(charles,harry).
mother(diana,william).
mother(diana,harry).
```

Diese Regeln definieren die Relation `mother` und `father`. Es handelt sich hier um *Fakten*, die ohne Vorbedingungen gelten sollen.

Die folgenden Regeln definieren die Prädikate `parent` und `grandma` auf Basis dieser Prädikate.

```
parent(P,C)    :- father(P,C); mother(P,C).
grandma(GM,GC) :- mother(GM,P), parent(P,GC).
```

Solche Regeln werden *Klauseln* genannt. Die Symbole in den Klauseln sind wie folgt zu interpretieren:

- `P :- Q` entspricht "$P \leftarrow Q$."
- `P , Q` entspricht "$P \wedge Q$".
- `P :- Q ; R` kürzt zwei Klauseln `P :- Q` und `P :- R` mit gleicher linker Seite ab.

Die beiden Klauseln sind also so zu lesen:

1. , "X ist Elternteil von Y, wenn X Mutter von Y ist, oder wenn X Vater von Y ist."
2. "X ist Großmutter von Y, wenn X Mutter von einer Person Z ist, und wenn diese Person Elternteil von Y ist."

Fakten und Klauseln bilden das logische Programm, das oft auch die *Datenbasis* genannt wird.

Fragen an die Datenbasis können nun wie folgt gestellt werden:

- Ist Elisabeth die Großmutter von William?

```
grandma(elizabethII,william)
⇝ mother(elizabethII,Z), parent(Z,william)      (Z ↦ Charles)
⇝ parent(charles,william)
⇝ father(charles,william) ⇝ yes
```

Zum "Beweis" dieser Anfrage werden die Fakten und Regeln der Datenbasis der Reihe nach ausprobiert, und wenn sie alle eliminiert werden können, gibt es – wie hier – die Antwort *yes*.
Diese Art der Auswertung ist ganz ähnlich wie die Auswertung einer booleschen Funktion in einer funktionalen Sprache. Die Frage, ob *Anne* die Großmutter von *William* sei, wird dagegen verneint:

```
grandma(anne,william)
⇝ mother(anne,Z), parent(Z,william)
⇝ Nein
```

Anne ist kinderlos, so dass die Anfrage `mother(anne,Z)` *fehlschlägt.*

- In einer Anfrage dürfen – wie erwähnt – aber auch *Unbekannte* (*Variablen*) auftreten, die im Laufe des "Beweises" gesetzt werden und als Erläuterung zu einer positiven Antwort angegeben werden.
  Die Anfrage "Wessen Großmutter ist Elizabeth II?" kann wie folgt formuliert und bearbeitet werden:

  `grandma(elizabethII,X)`
  $\rightsquigarrow$ `mother(elizabethII,Z), parent(Z,X)` (Z $\mapsto$ Charles) (1)
  $\rightsquigarrow$ `parent(charles,X)` (2)
  $\rightsquigarrow$ `father(charles,X)`
  $\rightsquigarrow$ *yes*, `X = william;`
  $\rightsquigarrow$ *yes*, `X = harry`

  Hierbei wird zunächst nur eine Antwort (X=william) geliefert, und auf Nachfrage weitere (`X=harry`). Im entscheidenden Schritt (1) wird die Anfrage `mother(elizabethII,Z)` mit der linken Seite der `grandma`-Regel zur Übereinstimmung gebracht (*unifiziert*), indem `X` auf `elizabethII` und `Y` auf `Z` gesetzt wird, auch auf der rechten Seite der Regel.
  Um zu (2) zu kommen, wird `mother(elizabethII,`*Z*`)` mit dem Fakt `mother(elizabethII,charles)` unifiziert, wodurch `Z` in der verbleibenden Anfrage auf `charles` gesetzt wird. Auf diese Anfrage passen zwei Fakten, wenn `X` auf `william` bzw. auf `harry` gesetzt wird.
- Man könnte auch umgekehrt, nach der Großmutter von William fragen:

  `grandma(X,william)`
  $\rightsquigarrow$ `mother(X,Z), father(Z,william)`
  $\rightsquigarrow \cdots \rightsquigarrow$
  $\rightsquigarrow$ `father(charles,william)`
  $\rightsquigarrow$ *yes*, `X = elizabethII`

  Hier gibt es nur eine Antwort, weil die Datenbasis Williams Großmutter mütterlicherseits nicht angibt. Dies nennt sich die *Annahme der vollständigen Welt* (*closed world assumption*): Wir müssen darauf vertrauen, dass die Datenbasis das relevante Wissen vollständig enthält.
  Hinter den $\cdots$ verbergen sich fehlschlagende Versuche, zu beweisen, dass `queenMum` die Großmutter von William sei, weil die Regeln und Fakten stur der Reihe nach abgearbeitet werden. Dies führt in eine Sackgase, weil wir kein Faktum finden, dass Philip als Vater von William ausgibt.
- Wir können noch allgemeiner mit '`grandma(X,Y)` nach der gesamten Großmutter-Relation fragen und bekommen dann nacheinander alle 6 G9roßmütter-EnkelInnen-Paare geliefert, die sich aus der Datenbasis ableiten lassen, von `X=queenMum, Y=anne` bis `X=elizabethII, Y=Harry`.

4.3.6 Logische Sprachen

Prolog ist die Mutter aller logischen Programmiersprachen und immer noch am weitesten bekannt. Spätere Versionen von Prolog haben die bei der Unifikation auf den *occur check* verzichtet. Das Ergebnis einer Unifikation kann dann ein unendlicher Term sein (dargestellt durch einen zyklischen Termgraphen).

In Prolog sind Zahlen so vordefiniert, dass die Operationen auf Zahlen wie in gewöhnlichen Programmiersprachen funktionieren: Der Wert einer Zahlen-Variable ist entweder gar nicht definiert, oder als eine konkrete Zahl. Andere Sprachen haben erlaubt, dass auch mit Zahlenvariablen und Operationen auf Zahlen unifiziert werden kann. Dann kann der Wert einer Zahlenvariablen auch durch eine Menge von Gleichungen und Ungleichungen definiert sein, also eine Teilmenge der Zahlen bestimmen. Dies Art von Unifikation wird *constraint solving* genannt.

Andere Sprachen wie Mercury [HCSJ96] haben Prolog um Typen und Module erweitert.

In Prolog kann nur mit Prädikaten eine Berechnung spezifiziert werden. Alle Funktionssymbole und Konstantennamen werden als freie Datenkonstruktoren aufgefasst, wie die Wertkonstruktoren in funktionalen Sprachen. Logisch-funktionale Sprachen wie Curry [Cur06] erlauben sowohl die Definition von Prädikaten als auch die Definition von Funktionen. Auf der theoretischen Ebene werden also Resolution und Termersetzung zu *Narrowing* kombiniert. Das bedeutet, dass die Regel (match) für Unifikation derart modifiziert wird, dass auf die linke oder rechte Seite Termersetzungsregeln angewendet werden dürfen, um die Terme gleich zu machen. Meistens werden Termersetzungsschritte nur dann gemacht, wenn eine Unifikation anders nicht möglich ist, und es werden nur so wenig solcher Schritte wie möglich gemacht.

Literaturhinweise

Als Zusatzquellen für dieses Kapitel benutzen wir

- JW. Klop und R de Vrijer: *First-Order Term Rewriting Systems* [KdV03]
- JW. Klop: *Term Rewriting Systems* [Klo92].
- U. Nilsson und J. Małuszyński: *Logic, Programming, and Prolog* [NM95]

Der Abschnitt über Termination gibt nur einen kurzen Einblick in das Buchkapitel [Zan03].