Logische Programmierung

Skriptum zur gleichnamigen Vorlesung im Sommersemester 1993

Prof. Dr. Gert Smolka
Dipl.–Inform. Christian Schulte

Inhaltsverzeichnis

Teil I

Grundlagen von Prolog

Kapitel 1

Uniforme Syntax

Wir werden in diesem Abschnitt einen Rahmen angeben, der es uns erlauben wird, formal exakte Definitionen auch für komplizierte Ableitungssysteme anzugeben.

1.1 Ordnungssortierte Syntax

Die Grundlage für eine syntaktische Konstruktion bildet wie immer die Vereinbarung, welche Symbole wie zur Konstruktion von syntaktischen Objekten herangezogen werden. Diese Grundlage liefert der folgende Begriff.

Definition 1.1 (Ordnungssortierte Signatur) *Eine* **ordnungssortierte Signatur** Σ *ist ein Quadrupel* $(SOR, \leq, OP, AR)$ *so daß:*

- SOR *ist eine nichtleere Menge, deren Elemente heißen* **Sorten**.
- $\leq$ *ist eine partielle Ordnung (d.h.* $\leq$ *ist eine reflexiv, antisymmetrische und transitive Relation) auf der Menge* SOR *der Sorten.*
- OP *ist eine nichtleere Menge, die Elemente heißen* **Operatoren**.
- $AR \subseteq OP \times SOR^+$*, Elemente dieser Relation heißen* **Aritäten**. *Ein Element* $(f, (A_1, \ldots, A_n, B)) \in AR$ *schreiben wir in der leichter lesbaren Form* $f : A_1 \times \cdots \times A_n \to B$.

Eine Signatur bestimmt somit die Grundelemente der Konstruktionen die wir vornehmen wollen. Jetzt wollen wir uns den eigentlichen syntaktischen Objekten zuwenden, denen ja unser Hauptinteresse gilt.

Definition 1.2 (Sortierte Terme) *Für eine ordnungssortierte Signatur* Σ *ist die Familie* $(TER_n(\Sigma, A))_{n \geq 0, A \in SOR}$ *von Mengen wie folgt definiert:*

$$\begin{aligned} TER_0(\Sigma, A) &:= \{f \in OP \mid \exists B : \; f : B \in AR \text{ und } B \leq A\} \\ TER_{n+1}(\Sigma, A) &:= TER_n(\Sigma, A) \end{aligned}$$

$$\begin{array}{lll} \cup & \{(f, s_1, \ldots, s_m) \mid & \exists A_1, \ldots, A_m, B \in SOR : \qquad \} \\ & & f : A_1 \times \cdots \times A_m \to B \in AR, \\ & & B \leq A, \\ & & s_1 \in TER_n(\Sigma, A_1) \\ & & \vdots \\ & & s_m \in TER_n(\Sigma, A_m) \end{array}$$

Die **Menge der sortierten Σ–Terme der Sorte** $A \in SOR$ *ist wie folgt definiert:*

$$TER(\Sigma, A) := \bigcup_{n \geq 0} TER_n(\Sigma, A).$$

Die **Menge der sortierten Σ–Terme** *ist wie folgt definiert:*

$$TER(\Sigma) := \bigcup_{A \in SOR} TER(\Sigma, A).$$

Für den sortierten Σ–Term $(f, s_1, \ldots, s_n)$ schreiben wir auch einfacher $f(s_1, \ldots, s_n)$.

1.2 Ordnungssortierte Kongruenzen

Die obigen Definitionen geben uns jetzt die syntaktischen Objekte, über die wir in Zukunft reden wollen. Häufig werden uns jedoch Situationen begegnen, in denen wir von der starren syntaktischen Struktur abweichen wollen. Das geeignete Mittel hierfür werden **Kongruenzen** sein, das heißt Äquivalenzrelationen, die verträglich bezüglich der syntaktischen Konstruktion sind.

Definition 1.3 (Σ–Kongruenz) *Eine* **Σ–Kongruenz** *bezüglich einer ordnungssortierten Signatur Σ ist eine Äquivalenzrelation $\equiv$ (d.h. eine reflexiv, symmetrische und transitive Relation) auf $TER(\Sigma)$ so daß gilt:*

Wenn

$$f(s_1, \ldots, s_n), f(t_1, \ldots, t_n) \in TER(\Sigma)$$

und

$$s_i \equiv t_i \qquad (1 \leq i \leq n),$$

dann auch

$$f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n).$$

Die generelle Vorgehensweise wird dabei die folgende sein: Wir definieren eine Relation M, die bestimmte Aussagen über unsere syntaktischen Objekte macht. Was wir aber dann in Wirklichkeit haben wollen, um bequem mit dieser Relation umzugehen, ist die kleinste Relation, die M umfasst, und die eine Kongruenz ist. Das dieses ein generell zulässiges Konstruktionsprinzip sagt uns die folgende Proposition.

Proposition 1.1 *Sei Σ eine ordnungssortierte Signatur und $Ax \subseteq TER(\Sigma) \times TER(\Sigma)$. Dann gibt es eine kleinste Σ–Kongruenz $\equiv$, so daß $Ax \subseteq \equiv$.*

1.3 Ordnungssortierte Kalküle

Bis jetzt haben wir nur über Syntax und Methoden gesprochen, die es uns erlauben etwas flexibler mit dieser umzugehen. Worauf wir in Zukunft jedoch unser Hauptaugenmerk richten werden sind **Kalküle**. Diese Kalküle werden gerade auf den vorher eingeführten Prinzipien aufbauen.

Definition 1.4 (Kalkül) *Ein* **Kalkül** *ist ein Tripel* $(\Sigma, \equiv, \longrightarrow)$ *wobei folgendes gilt:*

- Σ *ist eine ordnungssortierte Signatur.*
- $\equiv$ *ist eine* Σ*–Kongruenz.*
- $\longrightarrow \subseteq TER(\Sigma) \times TER(\Sigma)$ *ist eine Relation, die* **Ableitungsrelation** *heißt, und für die gilt:*
$$\equiv \circ \longrightarrow \circ \equiv \quad \subseteq \quad \longrightarrow .$$

Mit $\longrightarrow^*$ bezeichnen wir die kleinste transitive Relation, die $\equiv \cup \longrightarrow$ umfaßt[1].

1.4 Ordnungssortierte Syntax der Prädikatenlogik

Als Beispiel betrachten wir eine vereinfachte Version der Syntax der Prädikatenlogik bezüglich einer geeigneten ordnungssortierten Signatur.

Zunächst legen wir die Sorten unserer ordnungssortierten Signatur fest. Betrachten wollen wir Variablen, Konstanten, Terme, atomare Formeln und Formeln. Damit erhalten wir also:
$$SOR := \{\mathsf{Var}, \mathsf{Konst}, \mathsf{Term}, \mathsf{AFor}, \mathsf{For}\}.$$

Nun müssen wir die Relation $\leq$ zwischen den einzelnen Sorten festlegen. Unsere Idee ist die folgende:

- Jede Variable soll ein Term sein.
- Jede Konstante soll ein Term sein.
- Jede atomare Formel soll auch eine Formel sein.
- Es gibt keine anderen Beziehungen zwischen den Sorten SOR.

Damit kommen wir dann zu der folgenden Relation $\leq$:

$$\begin{array}{rcl} \leq & := & \{(\mathsf{Var}, \mathsf{Term}), (\mathsf{Konst}, \mathsf{Term}), (\mathsf{AFor}, \mathsf{For})\} \\ & \cup & \{(A, A) \mid A \in SOR\} \end{array}$$

[1] Man beachte den Unterschied zur transitiven und reflexiven Hülle, die normalerweise so bezeichnet wird. Hier gilt nicht nur Reflexivität, sondern allgemeiner $\equiv$.

Man beachte, daß diese Relation wirklich reflexiv (nach Konstruktion), antisymmetrisch und transitiv ist. In Zukunft werden wir nur noch das folgende angeben:

$$\begin{array}{rcl} \mathsf{Var} & \le & \mathsf{Term} \\ \mathsf{Konst} & \le & \mathsf{Term} \\ \mathsf{AFor} & \le & \mathsf{For} \end{array}$$

Diese Angabe soll dann gleichzeitig bedeuten, daß $\le$ die kleinste reflexive, antisymmetrische und transitive Relation ist, die zusätzlich die spezifizierten Paare enthält. Äquivalent ist, daß man die reflexive und transitive Hülle der angegebenen Paare bildet.

Nun wird es darum gehen, die Menge der Operatoren festzulegen. Wir werden jeweils nur endlich viele Variablen, Konstanten, Funktionszeichen und Prädikatszeichen vorsehen. Also definieren wir wie folgt:

$$\begin{array}{rcccccc} OP & := & \{\mathsf{X},\mathsf{Y},\mathsf{Z}\} & \cup & \{\mathsf{a},\mathsf{b},\mathsf{c}\} & \cup & \{\mathsf{f},\mathsf{g}\} \\ & \cup & \{\mathsf{p},\mathsf{q}\} & \cup & \{\wedge,\neg,\exists\} & & \end{array}$$

Neben den Operatoren müssen wir jetzt die Aritäten dieser Operatoren spezifizieren:

$$\begin{array}{lll} \mathsf{X}:\mathsf{Var} & \mathsf{a}:\mathsf{Konst} & \\ \mathsf{Y}:\mathsf{Var} & \mathsf{b}:\mathsf{Konst} & \mathsf{f}:\mathsf{Term}\to\mathsf{Term} \\ \mathsf{Z}:\mathsf{Var} & \mathsf{c}:\mathsf{Konst} & \mathsf{g}:\mathsf{Term}\times\mathsf{Term}\to\mathsf{Term} \end{array}$$

$$\begin{array}{ll} & \wedge:\mathsf{For}\times\mathsf{For}\to\mathsf{For} \\ \mathsf{p}:\mathsf{Term}\to\mathsf{AFor} & \neg:\mathsf{For}\to\mathsf{For} \\ \mathsf{q}:\mathsf{Term}\times\mathsf{Term}\to\mathsf{AFor} & \exists:\mathsf{Var}\times\mathsf{For}\to\mathsf{For} \end{array}$$

Ein Term über dieser nun vollständigen spezifizierten Signatur ist zum Beispiel:

$$\wedge\,(\exists\,(\mathsf{X},\mathsf{p}(\mathsf{X}))\,,\mathsf{q}\,(\mathsf{g}\,(\mathsf{X},\mathsf{f}\,(\mathsf{b}))))$$

Eine Konvention, der wir bereitwillig folgen, ist daß wir manche Operatoren in Infix–Notation oder Präfix–Notation schreiben. Wenn wir die obige Formel nach den üblichen Konventionen für $\wedge$ und $\exists$ schreiben, haben wir:

$$\exists\mathsf{X}\ (\mathsf{p}(\mathsf{X})\wedge\mathsf{q}\,(\mathsf{g}\,(\mathsf{X},\mathsf{f}(\mathsf{b}))))\,.$$

Wie wir an diesem Beispiel sehen, ist es relativ langwierig, alle Komponenten einer ordnungssortierten Signatur anzugeben. Aus diesem Grunde werden wir die gesamte Konstruktion vereinfachen. Das Beispiel sähe dann so aus:

$$\begin{array}{rcl} V & ::= & \mathsf{X} \mid \mathsf{Y} \mid \mathsf{Z} \\ K & ::= & \mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \\ S,T & ::= & V \mid K \mid \mathsf{f}(T) \mid \mathsf{g}(T,T) \\ A & ::= & \mathsf{p}(T) \mid \mathsf{q}(T,T) \\ F & ::= & A \mid F\wedge F \mid \neg F \mid \exists V(F) \end{array}$$

Die Korrespondenz ist dabei die folgende:

- Anstatt Sorten haben wir Metavariablen, die die Elemente dieser Sorte bezeichnen.
- Die Untersortenrelation $\leq$ ist implizit durch die Konstruktionsvorschriften gegeben, das heißt eine Produktion vom Typ $A ::= B$ bedeutet gleichzeitig $B \leq A$.
- Die Aritäten sind in gleicher Weise den Produktionen zu entnehmen.

Häufig werden wir die Menge aller Terme benennen wollen, die von einer Metavariablen bezeichnet werden. Dafür führen wir die Konvention ein, für die Menge aller Terme, die die Metavariable X bezeichnen kann, einfach $[X]$ zu schreiben.

Eine prädikatenlogische Signatur kann als Abkürzung für die entsprechende ordnungssortierte Signatur gesehen werden. Dabei sind die logischen Symbole $(\wedge, \neg, \exists, \ldots)$ und die Variablen a prori festgelegt.

Kapitel 2

Bäume als logische Datenstrukturen

Als Einstieg werden wir uns mit einer **Datenstruktur** für das logische Programmieren beschäftigen. Diese Datenstruktur wird nur kurz umrissen, da wir uns in Teil II noch intensiver damit beschäftigen werden.

2.1 Bäume

In diesem Abschnitt wollen wir eine formale Definition von **Bäumen** entwickeln.

Den Ausgangspunkt liefert uns eine beliebige Menge M. Wir konstruieren alle mindestens einstelligen Tupel über dieser Menge und fassen diese in der Menge M^+ zusammen. Formal[1] heißt das:

$$M^+ := \bigcup_{n \geq 1} M^n.$$

Wir konstruieren Bäume rekursiv, indem wir eine Marke und eine geordnete Sequenz von Unterbäumen haben. Die geordnete Sequenz ist also gerade wieder ein Tupel von Bäumen, und dieses Tupel können wir mit der Marke dadurch versehen, daß wir ein Paar (also ein 2–Tupel) aus dieser Marke und diesem Tupel konstruieren. Dieses führt uns zu der folgenden Definition.

Definition 2.1 (Bäume über M) *Sei M eine beliebige Menge. Wir definieren die Mengen $B_n[M]$ für alle $n \in \mathbb{N}_0$ wie folgt, dabei setzen wir jedoch voraus, daß die Mengen M und $B_n[M]$ disjunkt sind für alle $n \geq 0$.*

$$\begin{aligned} B_0[M] &:= M \\ B_{n+1}[M] &:= B_n[M] \cup \left(M \times B_n[M]^+\right) \end{aligned}$$

Die Elemente $B_n[M]$ sind **Bäume der Höhe** $\leq n$*, deren Blätter und Knotenmarkierungen aus der Menge M stammen.*

Die **Menge der Bäume über** M *wird mit $B[M]$ bezeichnet, und ist definiert als*

$$B[M] := \bigcup_{n \geq 0} B_n[M].$$

[1] Zur Erinnerung: M^n bezeichnet das n-fache kartesische Produkt über der Menge M.

Für einen Baum $(a, (\sigma_1, \ldots, \sigma_n)) \in B[M]$ schreiben wir in Zukunft einfach kürzer $a(\sigma_1, \ldots, \sigma_n)$. Syntaxdefinitionen werden wir immer in der folgenden Notation[2] angeben:

$a \in M$			
σ, τ	$::=$	$a(\sigma_1, \ldots, \sigma_n)$	$(n \geq 0)$

Diese Syntaxdefinition meint genau das folgende:

- a ist eine Metavariable, die immer Elemente von M bezeichnet.
- σ und τ sind Metavariablen, die immer Elemente von $B[M]$ denotieren.

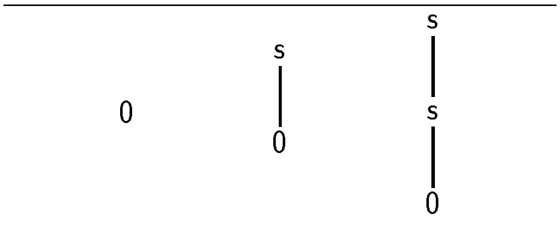
Wenn $\mathcal{M}$ eine Metavariable ist, so folgen wir der Konvention, daß die Metavariablen $\mathcal{M}'$, $\mathcal{M}''$, $\mathcal{M}'''$, $\mathcal{M}_1$, $\mathcal{M}_2$ und so fort Objekte der selben Klasse wie die Metavariable $\mathcal{M}$ beschreiben. Also in der obigen Definition haben wir davon Gebrauch gemacht, indem wir durch die Metavariablen σ_i auch Elemente von $B[M]$ bezeichnet haben.

Grundlegend für die Definition von Bäumen ist die Menge M. Im folgenden wollen wir eine Menge fixieren und dann Bäume über dieser Menge betrachten. Die Elemente dieser Menge heißen **Atome**, die Menge wollen wir mit *ATOM* bezeichnen. Diese Menge setzen wir als abzählbar unendlich voraus, die Buchstaben a, b und c werden im weiteren Elemente dieser Menge bezeichen.

Elemente der Menge $B[ATOM]$ werden wir im folgenden einfach mit den Metavariablen σ und τ bezeichen.

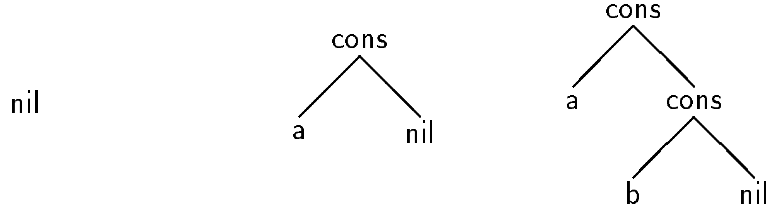
Wir wollen jetzt ein paar Beispiele für Bäume angeben. Bei diesen und auch weiteren Beispielen werden wir dieselbe konkrete Syntax für Atome verwenden, wie es auch die logische Programmiersprache PROLOG tut. Dabei besteht *ATOM* aus Bezeichnern, die mit einem Kleinbuchstaben anfangen, auf den dann Buchstaben, Ziffern oder das Zeichen _ folgen können. Beispiele für Atome sind also `datum`, `daTuM` und `datum_123_ABC`.

In bekannter Weise können wir natürliche Zahlen als Bäume darstellen, der Baum `0` beschreibt dabei die Zahl 0, die Bäume `s(0)` und `s(s(0))` die Zahlen 1 und 2. Graphisch können wir diese Bäume wie folgt darstellen:

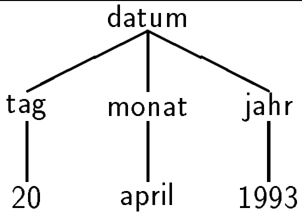


[2] Die Idee ist, Produktionen in erweiterter Backus-Naur Form (EBNF) anzugeben.

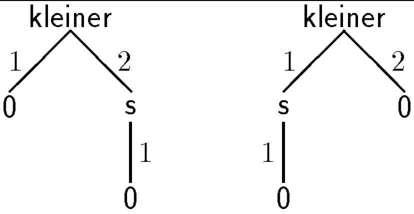
Listen können wir beispielsweise in der folgenden Art und Weise als Bäume darstellen:



Auch kompliziertere Datenstrukturen sind möglich. Ein Baum der das Datum vom 20. April 1993 darstellt, ist $\mathsf{datum(tag(20), monat(april), jahr(1993))}$, oder graphisch:



Wichtig ist jedoch, daß Bäume **geordnet** sind, das heißt, die Unterbäume sind geordnet. Also sind die beiden Bäume $\mathsf{kleiner(0, s(0))}$ und $\mathsf{kleiner(s(0), 0)}$ unterschiedlich. Eine genauere graphische Darstellung für diese beiden Bäume ist demnach:



Nimmt man die letzte Darstellung der beiden Bäume, so wird klar, daß man Bäume als markierte Tupel auffassen kann, wobei die Marken gerade Elemente von *ATOM* sind.

2.2 Bäume als Struktur $\mathcal{H}$

Wie man sich erinnert, denotiert eine Variable in einer logischen Formel ein Element aus dem Universum einer Struktur. Eine Formel beschreibt damit mögliche Werte für die Variablen dieser Formel. Beispiele hierfür sind die folgenden in der im Anhang A definierten Struktur $\mathcal{Z}$ der ganzen Zahlen gültigen Formeln, die diese Idee explizit machen:

$$\mathcal{Z} \models (\mathsf{X} - 2) * (\mathsf{X} + 3) \doteq 0 \leftrightarrow \mathsf{X} \doteq 2 \vee \mathsf{X} \doteq -3$$

oder

$$\mathcal{Z} \models (\mathsf{X} - 2)^3 \doteq 0 \leftrightarrow \mathsf{X} \doteq 2.$$

Gerade die erste Formel ist interessant, da diese **partielle** Information über die Werte der Variablen X ausdrückt.

In diesem Abschnitt wollen wir diese Ideen aus der Logik auf die vorhin eingeführten Bäume anwenden. Zu diesem Zweck werden wir jetzt die Struktur $\mathcal{H}$ einführen, die es uns erlauben wird, Bäume als eine **logische Datenstruktur** mit **logischen Variablen** zu benutzen.

Bevor wir diese Struktur angeben, legen wir zuerst eine passende Menge FUN von Funktionssymbolen fest, aus der wir dann eine Teilmenge als Signatur Σ auswählen können. FUN definieren wir wie folgt:

$$FUN := \{\mathsf{fun}(a, n) \mid a \in ATOM, n \in \mathbb{N}_0\}.$$

Man beachte, daß die Elemente von FUN jetzt gerade wieder spezielle Bäume sind. Die Stelligkeitsfunktion für ein Symbol ist wie folgt definiert:

$$|\mathsf{fun}(a, n)| := n.$$

Damit können wir ein- und dasselbe Atom a mit jeweils unterschiedlichen Stelligkeiten als Funktionszeichen verwenden. Prädikatszeichen werden wir später in ähnlicher Art und Weise festlegen.

Definition 2.2 (Die Struktur $\mathcal{H}$) *Die Struktur $\mathcal{H}$ besteht aus den folgenden Bestimmungsstücken:*

Signatur:	$\mathsf{fun}(a, n)$		$(n \in \mathbb{N}_0)$
Universum:	$[\sigma]$		
Interpretation:	$\mathsf{fun}(a, n)^{\mathcal{H}}(\sigma_1, \ldots, \sigma_n)$	$:=$	$a(\sigma_1, \ldots, \sigma_n)$

2.3 Gleichungssysteme

Mit Hilfe der Struktur $\mathcal{H}$ können wir Bäume durch Gleichungen beschreiben. Zum Beispiel können wir den Baum $\sigma = \mathsf{cons}(\mathsf{0}, \mathsf{nil})$ folgendermaßen beschreiben:

$$\alpha(\mathsf{X}) = \sigma \iff \mathcal{H}, \alpha \models \mathsf{X} \doteq \mathsf{fun}(\mathsf{cons}, 2)\Big(\mathsf{fun}(\mathsf{0}, 0), \mathsf{fun}(\mathsf{nil}, 0)\Big).$$

Diese etwas umständliche Notation werden wir dadurch vereinfachen, indem wir nur die Atome der Funktionszeichen angeben. Damit lautet die obige Aussage:

$$\alpha(\mathsf{X}) = \sigma \iff \mathcal{H}, \alpha \models \mathsf{X} \doteq \mathsf{cons}\,(\mathsf{0}, \mathsf{nil})\,.$$

In dem obigen Beispiel gab es nur genau einen Baum, der durch die Gleichung beschrieben wurde. Ein äußerst mächtiger Ansatz ist, die Beschreibung der Bäume unvollständig zu lassen, das heißt, ganze Klassen von Bäumen zu beschreiben. Wenn wir zum Beispiel die

Gleichungen

$$e \quad ::= \quad s \doteq t$$

Gleichungssysteme

$$\begin{array}{lcl} E, F & ::= & e \\ & \mid & E \wedge F \\ & \mid & \top \end{array}$$

Kongruenz

$$\begin{array}{lcll} E \wedge (E' \wedge E'') & \equiv & (E \wedge E') \wedge E'' & \textit{Assoziativität} \\ E \wedge E' & \equiv & E' \wedge E & \textit{Kommutativität} \\ E \wedge \top & \equiv & E & \textit{Identität} \end{array}$$

Abbildung 2.1: Gleichungssysteme

zweielementige Liste beschreiben wollen, wobei zusätzliche beide Elemente dieser Liste gleich sein sollen, so können wir dieses durch die folgende Formel tun:

$$\varphi = \exists \mathsf{Y}\, \mathsf{X} \doteq \mathsf{cons}(\mathsf{Y}, \mathsf{cons}(\mathsf{Y}, \mathsf{nil})).$$

Damit gilt dann:

$$\mathcal{H}, \alpha \models \varphi \iff \alpha(\mathsf{X}) \in \{\mathsf{cons}(\sigma, \mathsf{cons}(\sigma, \mathsf{nil})) \mid \sigma \text{ beliebiger Baum}\}.$$

Im weiteren wollen wir uns mit Gleichungen und Gleichungssystemen beschäftigen. Dazu definieren wir eine ordnungssortierte Signatur und Kongruenz in der im Abschnitt 1.4 eingeführten Kurzform, die Definition ist in Abbildung 2.1 dargestellt.

Von Interesse bei Gleichungssystemen sind für uns die enthaltenen Gleichungen, nicht jedoch in welcher Reihenfolge oder Klammerung diese Gleichungen in dem System auftreten. Anders gesagt, was uns eigentlich interessiert, sind **Multimengen** von Gleichungen. Um einerseits die erwünschten Eigenschaften von Multimengen zu haben, andererseits jedoch logische Formeln beizubehalten, unterlegen wir Gleichungssysteme der obigen Kongruenzrelation, die die Eigenschaften von Multimengen besitzt.

Definition 2.3 *Wir wollen ein Gleichungssystem E* **unifizierbar** *nennen, wenn E erfüllbar in $\mathcal{H}$ ist, das heißt es gibt eine Belegung α nach $\mathcal{H}$, so daß: $\mathcal{H}, \alpha \models E$.*

Es ist natürlich klar, daß es Gleichungssysteme gibt, die nicht unifizierbar sind. Zum Beispiel sind die folgenden Gleichungssysteme nicht unifizierbar:

- $\mathsf{X} \doteq \mathsf{a(Z)} \wedge \mathsf{X} \doteq \mathsf{b(Y)}$, denn eine Lösung von X kann nicht sowohl mit dem Knoten a als auch mit dem Knoten b anfangen.

- $\mathsf{X} \doteq \mathsf{a(Y)} \wedge \mathsf{Y} \doteq \mathsf{b(X)}$, da jede Lösung ein unendlicher Baum sein müßte. Diese Art von Bäumen haben wir jedoch ausgeschlossen, werden uns aber später in Teil II noch näher damit beschäftigen.

Ein lineares Gleichungssystem in der linearen Algebra kann man durch das Gaußsche Eliminationsverfahren auf eine Form bringen, die es gestattet, seine Lösungen direkt abzulesen. Wir werden nun sehen, daß es auch bei unseren Gleichungssystemen so eine Form gibt.

Definition 2.4 *Ein Gleichungssystem E ist* **gelöst**, *falls E die Form*

$$x_1 \doteq t_1 \wedge \cdots \wedge x_n \doteq t_n \qquad\qquad (n \geq 0)$$

hat, wobei die x_i paarweise verschieden sein müssen, und kein x_i in einem t_j vorkommen darf.

Einfach zu zeigen ist die folgende Proposition.

Proposition 2.1 *Eine gelöstes Gleichungssystem ist unifizierbar.*

Der folgende Satz, den wir später beweisen werden, sagt aus, daß zu jedem lösbaren Gleichungssystem auch eine gelöste Form existiert.

Satz 2.2 *Zu jedem unifizierbaren Gleichungssystem E existiert ein gelöstes Gleichungssystem F, so daß:*

$$E \models\!\mid_{\mathcal{H}} F \qquad \textit{und} \qquad \mathcal{V}(F) \subseteq \mathcal{V}(E).$$

Ein Beispiel für ein unifizierbares Gleichungssystem ist:

$$\mathsf{X} \doteq \mathsf{s(s(Y))} \wedge \mathsf{X} \doteq \mathsf{s(s(0))}$$

Eine gelöste Form hiervon ist:

$$\mathsf{X} \doteq \mathsf{s(s(0))} \wedge \mathsf{Y} \doteq \mathsf{0}$$

2.4 Substitutionen und Unifikatoren

Eine **Substitution** ist eine Abbildung von Variablen auf Terme. Wir werden Substitutionen mit θ oder ρ bezeichnen. Die Menge $\{x \in VAR \mid \theta(x) \neq x\}$ werden wir als **Definitionsbereich** oder $\mathcal{D}om(\theta)$, die Menge $\{\theta(x) \in T(\Sigma) \mid x \in \mathcal{D}om(\theta)\}$ als **Wertebereich** oder $\mathcal{R}an(\theta)$ bezeichnen.

Gegeben sei eine Substitution θ. Wir setzen diese Substitution zu einer Abbildung θ^* auf Termen und quantorenfreie Formeln fort:

$$\begin{array}{rcll}
\theta^*(x) & := & \theta(x) & \\
\theta^*\left(f(t_1,\ldots,t_n)\right) & := & f(\theta^*(t_1),\ldots,\theta^*(t_n)) & \\
\theta^*(\top) & := & \top & \\
\theta^*(\bot) & := & \bot & \\
\theta^*(s \doteq t) & := & \theta^*(s) \doteq \theta^*(t) & \\
\theta^*\left(p(t_1,\ldots,t_n)\right) & := & p(\theta^*(t_1),\ldots,\theta^*(t_n)) & \\
\theta^*\left(\neg\varphi\right) & := & \neg(\theta^*(\varphi)) & \\
\theta^*\left(\varphi \odot \psi\right) & := & \theta^*(\varphi) \odot \theta^*(\psi) & \qquad \odot \in \{\wedge,\vee,\rightarrow,\leftrightarrow\}
\end{array}$$

Der Einfachheit halber bezeichnen wir die Abbildung θ^* wieder mit θ. Die **Komposition** von zwei Substitutionen θ und ρ bezeichnen wir mit $\theta \circ \rho$, diese ist definiert als $(\theta \circ \rho)(x) := \theta(\rho(x))$.

Proposition 2.3 *Die Komposition von zwei Substitutionen ist wieder eine Substitution.*

Ist $\mathcal{D}om(\theta) = \{x_1,\ldots,x_n\}$, so heißt θ **endlich** und wir schreiben θ einfach als

$$[x_1/\theta(x_1),\ldots,x_n/\theta(x_n)].$$

Im weiteren werden wir oft Substitutionen $[x_1/t_1,\ldots,x_n/t_n]$ mit dem Gleichungssystem $x_1 \doteq t_1 \wedge \cdots \wedge x_n \doteq t_n$ identifizieren.

Definition 2.5 (Unifikator) *Sei $E = s_1 \doteq t_1 \wedge \cdots \wedge s_n \doteq t_n$ ein Gleichungssystem und θ eine endliche Substitution.*

θ heißt **Unifikator von** *E, wenn:*

$$\theta s_i = \theta t_i \qquad\qquad (1 \le i \le n)$$

θ heißt **prinzipaler Unifikator von** *E, wenn folgendes gilt:*

- *θ ist ein Unifikator.*
- *θ ist* **allgemeinst***: Wenn θ' ein Unifikator von E ist, so existiert eine Substitution ρ mit $\theta' = \rho \circ \theta$.*
- *θ ist* **idempotent***, das heißt $\theta \circ \theta = \theta$.*
- *$\mathcal{D}om(\theta) \cup \mathcal{V}(\mathcal{R}an(\theta)) \subseteq \mathcal{V}(E)$.*

Proposition 2.4 *Sei E ein Gleichungssystem und θ eine Substitution. Dann gelten die folgenden Aussagen:*

- *E ist genau dann unifizierbar, wenn es einen Unifikator für E gibt.*
- *θ ist genau dann ein Unifikator von E, wenn $\theta \models_{\mathcal{H}} E$.*

- *θ ist genau dann ein prinzipaler Unifikator von E, falls die beiden Aussagen gelten:*
    1. *θ ist gelöst und enthält nur Variablen aus $\mathcal{V}(E)$.*
    2. *$E \models\!|_{\mathcal{H}}\, \theta$.*

Definition 2.6 *Ein Term s (Eine quantorenfreie Formel φ) heißt* **Instanz** *des Terms t (der Formel ψ), falls es eine Substitution θ gibt, so daß $s = \theta(t)$ ($\varphi = \theta(\psi)$).*

Eine quantorenfreie Formel φ heißt **Variante von** *ψ, falls φ Instanz von ψ und ψ Instanz von φ ist.*

Kapitel 3

Berechnung

In diesem Kapitel werden wir uns mit den wichtigen Ideen des logischen Programmierens beschäftigen. Dies Idee ist, Logik als Programmiersprache und Deduktion als Berechnung zu benutzen.

3.1 Formeln als Programmiersprache

Wir werden nicht alle Arten von logischen Formeln als Programme zulassen, sondern wählen eine spezielle Klasse von Formeln aus. Bevor wir diese Klasse angeben legen wir die Menge der Relationszeichen fest:

$$PR\ddot{A} := \{\mathsf{rel}(a, n) \mid a \in ATOM, n \in \mathbb{N}_0\}.$$

Für diese Zeichen ist die Stelligkeitsfunktion wie folgt definiert:

$$|\mathsf{rel}(a, n)| := n.$$

Damit ergibt sich die Syntax zu:

Literale:	A, B	$::=$	$\mathsf{rel}(a, n)(t_1, \ldots, t_n)$
Ziele[1]*:*	G, H	$::=$	$A \quad \mid \quad \top \quad \mid \quad G \wedge H$
Klauseln:			$A \leftarrow G$
Programme:	P, Q	:	endliche Menge von Klauseln

Klauseln werden wir auch als **Hornklauseln**, Ziele auch als **Anfragen** bezeichnen. Bei einer Klausel $A \leftarrow G$ werden wir A als **Kopf der Klausel** und G als **Rumpf der Klausel** bezeichnen. Ein Literal $\mathsf{rel}(a, n)(t_1, \ldots, t_n)$ werden wir wieder kürzer als $a(t_1, \ldots, t_n)$ bezeichnen.

Wenn ein Programm P gegeben ist, so sind wir daran interessiert, an dieses Programm eine Anfrage G zu stellen, und erwarten als Berechnungsdienst das folgende:

[1] G leitet sich aus dem englischen Wort „goal“ für Ziel her.

- Ist G eine logische Konsequenz aus P?
- Wie müssen Belegungen für die freien Variablen in G gebaut sein, so daß G eine logische Konsequenz unter einer Belegung ist?
- Eventuelles Aufzählen aller Lösungen.

Das bedeutet, daß wir an der folgenden Menge von Antworten interessiert sind:

$$ANT_P(G) := \{\theta \mid P \models \theta G\}.$$

Bezeichnen wollen wir diese Menge als **Antworten von G bezüglich P.**

Trivialerweise gelten die folgenden Tatsachen:

- Wenn $P \models G$, dann gilt für alle Substitutionen θ: $P \models \theta G$.
- $ANT_P(G)$ ist unendlich.

Das erste Faktum läßt sich äquivalent auch so wiedergeben: Falls $\theta \in ANT_P(G)$, dann ist $\rho \circ \theta \in ANT_P(G)$. Das heißt, daß diese Menge unter Linkskomposition abgeschlossen ist. Damit drängt sich natürlich sofort die Frage auf, ob es eine besonders kleine Menge gibt, die die „gleiche" Information modulo Instantiierung wie $ANT_P(G)$ ausdrückt. Man denke nur an Vektorräume in der linearen Algebra, bei diesen spielen gerade die Basen (also die minimalen Erzeugendensysteme) bezüglich Linearkombination dieser Räume diese Rolle. Diese Überlegung führt uns zu der folgenden Definition:

Definition 3.1 (Vollständige Antwortmenge) *Eine Menge M von Substitutionen ist eine* **vollständige Antwortmenge** *für ein Ziel G bezüglich eines Programmes P, falls:*

1. *$M \subseteq ANT_P(G)$.*
2. *Für alle $\theta \in ANT_P(G)$ existiert ein $\rho \in M$, so daß θG Instanz von ρG ist.*

Beispiel 3.1 (Natürliche Zahlen) Sei P die folgende Menge von Klauseln:

$$\begin{aligned} \mathsf{nat(0)} &\leftarrow \top \\ \mathsf{nat(s(X))} &\leftarrow \mathsf{nat(X)} \end{aligned}$$

Die Menge der Antworten bezüglich der Anfrage $\mathsf{nat(X)}$ ist

$$ANT_P(\mathsf{nat(X)}) = \{\theta \mid \theta(\mathsf{X}) = \mathsf{s}^n(\mathsf{0}), n \in \mathbb{N}_0\}.$$

Eine vollständige Menge (es ist sogar die kleinste!) ist

$$\{[\mathsf{X}/\mathsf{s}^n(\mathsf{0})] \mid n \in \mathbb{N}_0\}.$$

□

Mit Hilfe dieser Definition sind wir auch im Stande, unsere Wünsche an den Berechnungsdienst zu konkretisieren: Wir wollen ein praktikables Verfahren, das bei gegebenem P und G ein vollständiges $M \subseteq ANT_P(G)$ enumeriert. Wünschenswert ist natürlich auch: Falls ein endliches M existiert, sollte beim Enumerieren irgendwann angezeigt werden „Es gibt keine weiteren Lösungen mehr". Leider ist dieses allgemein nicht möglich, da dies die Lösbarkeit des Halteproblems implizieren würde.

3.2 Herbrand–Strukturen

Bevor wir uns der operationalen Seite zuwenden, wollen wir noch eine andere Betrachtungsweise der deklarativen Semantik von Programmen vorstellen. Bisher haben wir immer Aussagen der Form

$$P \models \theta G$$

betrachtet. In dieser Aussage müssen wir alle Modelle von P betrachten, denn sie ist äquivalent zu:

$$\text{Für alle Modelle } \mathcal{A} \text{ von } P\text{:} \quad \mathcal{A} \models \theta G.$$

Wünschenswert wäre es, wenn es **ein** ausgezeichnetes Modell gäbe, das in einem gewissen Sinne die Bedeutung des Programms P angibt.

Dazu führen wir ein besonders einfache Art von Strukturen ein.

Definition 3.2 (Herbrand-Strukturen) *Eine Struktur $\mathcal{A}$ ist eine Herbrand–Struktur, falls:*

- $\mathcal{U}(\mathcal{A}) := \bigcup_{i \geq 0} TB_n$
- *Für jedes Funktionssymbol $f = \mathsf{fun}(a,n)$ ist $\mathcal{A}[\![f]\!](t_1,\ldots,t_n) = a(t_1,\ldots,t_n)$.*

Dabei sind die Mengen TB_i wie folgt definiert:

$$\begin{array}{rcl} TB_0 & := & FUN = \{\mathsf{fun}(a,n) \mid a \in ATOM, n \in \mathbb{N}_0\} \\ TB_{n+1} & := & TB_n \cup \{\mathsf{fun}(a,n)(t_1,\ldots,t_n) \mid \mathsf{fun}(a,n) \in FUN, t_i \in TB_n\} \end{array}$$

Bei dieser Definition ist zu beachten, daß nur noch die Interpretation der Prädikatszeichen zu wählen ist, alle anderen Bestimmungsstücke sind vorgegeben. Außerdem haben wir auch schon die Menge $PR\ddot{A}$ fixiert (siehe oben).

Obige Definition können wir auch so formulieren: $\mathcal{A}$ ist eine Herbrand-Struktur, falls Universum und Interpretation der Funktionszeichen in $\mathcal{A}$ mit der der Struktur $\mathcal{H}$ zusammenfallen.

Herbrand-Strukturen haben für uns sehr angenehme Eigenschaften. Wir wollen zunächst eine fast triviale Aussage formulieren.

Proposition 3.1 *Jedes Programm P hat eine Herbrand-Modell.*

Den Durchschnitt von Herbrand-Strukturen definieren wir wie folgt:

Definition 3.3 *Sei $(\mathcal{H}_i)_{i\in I}$ eine Familie von Herbrand-Strukturen, die alle bezüglich derselben Signatur definiert sein müssen. Den* **Durchschnitt** $\bigcap_{i\in I}\mathcal{H}_i$ *definieren wir als die folgende Herbrand-Struktur:*

$$\bigcap_{i\in I}\mathcal{H}_i\,[\![\mathsf{rel}(a,n)]\!] := \bigcap_{i\in I}\mathcal{H}_i\,[\![\mathsf{rel}(a,n)]\!]\,.$$

Man beachte, daß die obige Definition die Struktur $\bigcap_{i\in I}\mathcal{H}_i$ vollständig beschreibt: Dazu bedurfte es, wie weiter oben erwähnt, nur der Interpretation der Prädikatszeichen.

Lemma 3.2 *Sei P ein Programm und $(\mathcal{H}_i)_{i\in I}$ eine Familie von Herbrand-Modellen von P. Dann ist*

$$\bigcap_{i\in I}\mathcal{H}_i$$

auch ein Herbrand-Modell von P.

Damit erhalten wir eine interessante Aussage über die Existenz und das Aussehen eines „kanonischen“ Modells für ein Programm.

Satz 3.3 (Kleinstes Herbrand-Modell) *Sei P ein Programm und $(\mathcal{H}_i)_{i\in I}$ alle Herbrand-Modelle von P. Dann ist*

$$\mathcal{H}(P) := \bigcap_{i\in I}\mathcal{H}_i$$

das eindeutig bestimmte kleinste (bezüglich $\subseteq$) Herbrand-Modell von P.

Die abschließende Aussage ist gerade das Ziel, das wir erreichen wollten.

Lemma 3.4 *Sei P ein Programm und G eine Anfrage. Dann gilt:*

$$P \models \exists G \iff \mathcal{H}(P) \models \exists G.$$

Die hier eingeführten Begriffe werden uns im weiteren noch oftmals dienlich sein: Einerseits zur Verstärkung der Intuition (Wir wissen jetzt ja, wie einfach die deklarative Semantik eines Programmes ist!), andererseits als Hilfsmittel für weitere Konstruktionen.

3.3 Der Kalkül 1

Wir werden jetzt einen einfachen Kalkül, den Kalkül 1, entwerfen, der eine erste Approximation an das von uns gewünschte Berechnungsverfahren sein wird.

Der Kalkül 1 ist in Abbildung 3.1 definiert. Man beachte, daß die binäre Relation $\longrightarrow$ in Abhängigkeit von einem Programm P definiert ist.

Der deklarativen Charakterisierung der Menge der Antworten können wir jetzt eine operationale Charakterisierung bezüglich des Kalküls 1 entgegenstellen. Bevor wir dieses jedoch machen, werden wir noch einen technischen Begriff einführen.

Definition 3.4 *Ein Tripel $(P, G, \overline{x})$ heißt* **zulässig**, *falls folgendes gilt:*

- *Sowohl in P als auch in G kommt das Prädikatensymbol $\mathsf{rel}(@, n)$ für kein $n \in \mathbb{N}$ vor.*
- *$\overline{x}$ ist eine Folge paarweise verschiedener Variablen.*

Syntax

Literale:	A, B	$::=$	$a(t_1, \ldots, t_n)$
Ziele:	G, H	$::=$	$A \quad \mid \quad \top \quad \mid \quad G \wedge H$
Klauseln:			$A \leftarrow G$
Programme:	P, Q	:	endliche Menge von Klauseln

Kongruenz

Kommutativität:	$G_1 \wedge G_2$	$\equiv$	$G_2 \wedge G_1$
Assoziativität:	$G_1 \wedge (G_2 \wedge G_3)$	$\equiv$	$(G_1 \wedge G_2) \wedge G_3$
Identität:	$G \wedge \top$	$\equiv$	G

Reduktion

$G \longrightarrow H \wedge G'$ falls:

- $A \wedge G'$ Instanz von G;
- $A \leftarrow H$ Instanz einer Klausel aus P.

Abbildung 3.1: Der Kalkül 1

- $\mathcal{V}(\overline{x}) = \mathcal{V}(G)$.

Wenn ein Tripel $(P, G, \overline{x})$ zulässig ist, werden wir im weiteren kürzer sagen, daß P, G und $\overline{x}$ zulässig sind.

Definition 3.5 (Berechnete Antworten im Kalkül 1) *Seien P, G, $\overline{x}$ zulässig. Die* **Menge der berechneten Antworten** *von G bezüglich P ist wie folgt definiert:*

$$ANT_1(G, \overline{x}) = \{[\overline{x}/\overline{s}] \mid \exists \overline{s} :\ G \wedge @(\overline{x}) \longrightarrow^* @(\overline{s})\}.$$

Dieser Kalkül leistet im wesentlichen das, was wir von einem Berechnungsdienst gefordert haben. Dies sagt gerade der folgende Satz aus.

Satz 3.5 (Korrektheit und Vollständigkeit) *Seien P, G und $\overline{x}$ zulässig. Der Kalkül 1 ist korrekt und vollständig, genauer:*

$$ANT_1(G, \overline{x}) \textit{ ist vollständig für } ANT_P(G)$$

Beispiel 3.2 (Fortsetzung von Beispiel 3.1) Eine Ableitung von $\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{X})))$ im Kalkül 1 ist:

$$\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{X}))) \wedge @(\mathsf{X}) \quad \longrightarrow \quad \mathsf{nat}(\mathsf{s}(\mathsf{X})) \wedge @(\mathsf{X})$$

$$\begin{aligned} &\longrightarrow \mathsf{nat}(\mathsf{X}) \wedge @(\mathsf{X}) \\ &\longrightarrow \top \wedge @(\mathsf{0}) \\ &\equiv @(\mathsf{0}) \end{aligned} \tag{3.1}$$

Anstatt bei (3.1) die erste Klausel des Programms auszuwählen, wäre auch die zweite Klausel möglich gewesen, also:

$$\begin{aligned} (3.1) &\longrightarrow \mathsf{nat}(\mathsf{X}') \wedge @(\mathsf{s}(\mathsf{X}')) \\ &\longrightarrow \top \wedge @(\mathsf{s}(\mathsf{0})) \\ &\equiv @(\mathsf{s}(\mathsf{0})) \end{aligned}$$

□

Kalkül 1 weist folgende Indeterminismen auf:

- Auswahl des reduzierenden Literals (dies entspricht gerade der Kongruenz $\equiv$.)
- Auswahl der Instanz des Ziels.
- Auswahl der Instanz der Klausel.
- Auswahl der Klausel.

Alle Indeterminismen bis auf den letzten sind „don't care“, das heißt, wie man die Auswahl trifft, beeinflußt die Antwortmenge nicht. Dagegen ist die Auswahl der Klausel „don't know“: Würde hier nur deterministische Auswahl zugelassen, würde der Kalkül unvollständig werden.

3.4 Der Kalkül 2

Eine Quelle von Indeterminismus im Kalkül 1 war die beliebige Auswahl von Instanzen von Klauseln und Zielen. Diese beliebige Freiheit ist jedoch nicht unbedingt notwendig um eine vollständige Antwortmenge zu erhalten.

Beispiel 3.3 Sei P das Programm

$$\mathsf{gleich}(\mathsf{X},\mathsf{X}) \leftarrow \top$$

und G das Ziel:

$$\mathsf{gleich}(\mathsf{f}(\mathsf{g}(\mathsf{X}),\mathsf{Y},\mathsf{Z}),\mathsf{f}(\mathsf{Y},\mathsf{Y},\mathsf{c}))$$

Mit Hilfe des Kalküls 1 können wir die folgenden Antworten berechnen:

$$\begin{array}{l} [\mathsf{X}/\ \mathsf{c}\ ,\mathsf{Y}/\ \mathsf{g}(\mathsf{c})\ ,\mathsf{Z}/\mathsf{c}] \\ [\mathsf{X}/\ \mathsf{g}(\mathsf{c})\ ,\mathsf{Y}/\ \mathsf{g}(\mathsf{g}(\mathsf{c}))\ ,\mathsf{Z}/\mathsf{c}] \\ [\mathsf{X}/\mathsf{f}(\mathsf{c},\mathsf{c},\mathsf{g}(\mathsf{c})),\mathsf{Y}/\mathsf{g}(\mathsf{f}(\mathsf{c},\mathsf{c},\mathsf{g}(\mathsf{c}))),\mathsf{Z}/\mathsf{c}] \end{array}$$

Alle diese Antworten lassen sich jedoch aus der Antwort

$$[\mathsf{Y}/\mathsf{g}(\mathsf{X}),\mathsf{Z}/\mathsf{c}]$$

durch weitere Instantiierung von X erhalten. □

Syntax

Wie in Abbildung 3.1.

Kongruenz

Kommutativität:	$G_1 \wedge G_2$	$\equiv$	$G_2 \wedge G_1$
Assoziativität:	$G_1 \wedge (G_2 \wedge G_3)$	$\equiv$	$(G_1 \wedge G_2) \wedge G_3$
Identität:	$G \wedge \top$	$\equiv$	G

Reduktion

$A \wedge G \longrightarrow \theta H \wedge \theta G$ falls

- $B \leftarrow H$ Variante einer Klausel aus P;
- $B \leftarrow H$ variablendisjunkt zu $A \wedge G$;
- θ prinzipaler Unifikator von $A \doteq B$.

Abbildung 3.2: Der Kalkül 2

Wir suchen somit nach einem Kalkül der nur noch die letzte Antwort erzeugt, aber alle anderen Antworten unterdrückt. Dies wird gerade der Kalkül 2 leisten, er ist in Abbildung 3.2 definiert.

In Zukunft werden wir die Kongruenz $\equiv$ auf Zielen in der Kurzform: $([G], \wedge, \top)$ ist ein kommutatives Monoid beschreiben, und meinen damit, daß $\wedge$ kommutativ und assoziativ auf Zielen ist, und daß $\top$ neutrales Element bezüglich $\wedge$ ist.

Definition 3.6 (Berechnete Antworten im Kalkül 2) *Seien P, G und $\overline{x}$ zulässig. Die* **Menge der berechneten Antworten** *von G bezüglich P, ist wie folgt definiert:*

$$ANT_2(G, \overline{x}) = \{[\overline{x}/\overline{s}] \mid \exists \overline{s} : \ G \wedge @(\overline{x}) \longrightarrow^* @(\overline{s})\}.$$

Auch der Kalkül hat die grundlegenden Eigenschaften, die wir von einem Kalkül verlangen.

Satz 3.6 (Korrektheit und Vollständigkeit) *Seien P, G und $\overline{x}$ zulässig. Der Kalkül 2 ist korrekt und vollständig, genauer:*

$$ANT_2(G, \overline{x}) \textit{ ist vollständig für } ANT_P(G)$$

3.5 Der Kalkül 3

In einigen Punkten läßt der Kalkül 2 noch Wünsche offen. Ein triviales Beispiel dafür ist das folgende.

Beispiel 3.4 Sei P das folgende Programm

$$\{\mathsf{p(a)} \leftarrow \top, \mathsf{p(b)} \leftarrow \top, \mathsf{q} \leftarrow \mathsf{q}\}.$$

Es ist klar, daß sich das Ziel

$$\mathsf{q} \wedge \mathsf{p(c)} \wedge @$$

nicht auf @ reduzieren läßt, das heißt $ANT_P(\mathsf{q} \wedge \mathsf{p(c)}) = \emptyset$. Dennoch gibt es im Kalkül 2 nur die einzige (modulo $\equiv$) folgende unendliche Ableitung

$$\mathsf{q} \wedge \mathsf{p(c)} \wedge @ \longrightarrow \mathsf{q} \wedge \mathsf{p(c)} \wedge @ \longrightarrow \mathsf{q} \wedge \mathsf{p(c)} \wedge @ \longrightarrow \cdots$$

Ein etwas „schlauerer" Kalkül könnte jedoch feststellen, daß sich das Literal $\mathsf{p(c)}$ nie reduzieren läßt, und daß es somit keine erfolgreiche Ableitung geben kann. $\square$

Die Idee für einen „schlaueren" Kalkül wird sein, die Möglichkeiten für die Reduktion eines Literals explizit zu machen, und somit solche Fälle wie in dem obigen Beispiel zu erkennen.

Dafür müssen wir zu **Systemen von Zielen** und zu veränderten **Programmen** übergehen. Insgesamt erhalten wir dann den Kalkül 3, der in Abbildung 3.3 definiert ist.

Wir werden auch diesem Kalkül logische Aussagen gegenüberstellen, dabei werden wir jedoch die folgenden Konventionen implizit anwenden:

- Wir identifizieren Literale mit atomaren Formeln,
- Ziele und Klauseln mit den entsprechenden logischen Formeln, und
- Programme mit der sich ergebenden Menge von Klauseln.

Das obige Beispiel wollen wir jetzt im Kalkül 3 betrachten.

Beispiel 3.5 (Fortsetzung von Beispiel 3.4) In der neuen Syntax haben wir das folgende Programm

$$(\mathsf{p(a)} \leftarrow \top) \cdot (\mathsf{p(b)} \leftarrow \top) \cdot (\mathsf{q} \leftarrow \mathsf{q}).$$

Im Kalkül 3 können wir die folgende Ableitung herstellen:

$$\begin{array}{rcl}
\mathsf{q} \wedge \mathsf{p(c)} \wedge @ & \equiv & \mathsf{p(c)} \wedge \mathsf{q} \wedge @ \\
 & \equiv & (\mathsf{p(c)} \wedge \mathsf{q} \wedge @) \cdot \bot \\
 & \longrightarrow & \mathsf{res}(\mathsf{p(c)}, (\mathsf{p(a)} \leftarrow \top) \cdot (\mathsf{p(b)} \leftarrow \top) \cdot (\mathsf{q} \leftarrow \mathsf{q}), \mathsf{q} \wedge @) \cdot \bot \\
 & \longrightarrow & \mathsf{res}(\mathsf{p(c)}, (\mathsf{p(b)} \leftarrow \top) \cdot (\mathsf{q} \leftarrow \mathsf{q}), \mathsf{q} \wedge @) \cdot \bot \\
 & \longrightarrow & \mathsf{res}(\mathsf{p(c)}, (\mathsf{q} \leftarrow \mathsf{q}), \mathsf{q} \wedge @) \cdot \bot \\
 & \equiv & \mathsf{res}(\mathsf{p(c)}, (\mathsf{q} \leftarrow \mathsf{q}) \cdot \bot, \mathsf{q} \wedge @) \cdot \bot \\
 & \longrightarrow^* & \bot \cdot \bot \\
 & \equiv & \bot
\end{array}$$

Syntax

Literale:	A, B	$::=$	$a(t_1, \ldots, t_n)$
Ziele:	G, H	$::=$	$A \mid \top \mid G \wedge H$
Klauseln:			$A \leftarrow G$
Systeme:	S	$::=$	$G \mid \bot \mid \mathsf{res}(A, P, G) \mid S \cdot S$
Programme:	P, Q	$::=$	$A \leftarrow G \mid \bot \mid P \cdot Q$

Kongruenz

- $([G], \wedge, \top)$ ist kommutatives Monoid.
- $([P], \cdot, \bot)$ ist kommutatives Monoid.
- $([S], \cdot, \bot)$ ist kommutatives Monoid.

Reduktion

- $(A \wedge G) \cdot S \longrightarrow \mathsf{res}(A, Q, G) \cdot S$

  falls:
  - Q Variante von P;
  - Q variablendisjunkt zu $A \wedge G$;
  - $A \neq @(\cdots)$.
- $\mathsf{res}(A, \bot, G) \cdot S \longrightarrow \bot \cdot S$
- $\mathsf{res}(A, (B \leftarrow H) \cdot Q, G) \cdot S \longrightarrow \mathsf{res}(A, Q, G) \cdot S$

  falls A und B nicht unifizierbar.
- $\mathsf{res}(A, (B \leftarrow H) \cdot Q, G) \cdot S \longrightarrow (\theta H \wedge \theta G) \cdot \mathsf{res}(A, Q, G) \cdot S$

  falls θ prinzipaler Unifikator von $A \doteq B$.

Abbildung 3.3: Kalkül 3

Es ist jedoch auch weiterhin möglich, eine unendliche Ableitung herzustellen:

$$\begin{array}{rcl}
\mathsf{q} \wedge \mathsf{p(c)} \wedge @ & \equiv & (\mathsf{q} \wedge \mathsf{p(c)} \wedge @) \cdot \bot \\
& \longrightarrow & \mathsf{res}(\mathsf{q}, (\mathsf{p(a)} \leftarrow \top) \cdot (\mathsf{p(b)} \leftarrow \top) \cdot (\mathsf{q} \leftarrow \mathsf{q}), \mathsf{p(c)} \wedge @) \cdot \bot \\
& \equiv & \mathsf{res}(\mathsf{q}, (\mathsf{q} \leftarrow \mathsf{q}) \cdot (\mathsf{p(a)} \leftarrow \top) \cdot (\mathsf{p(b)} \leftarrow \top), \mathsf{p(c)} \wedge @) \cdot \bot \\
& \longrightarrow & (\mathsf{q} \wedge \mathsf{p(c)} \wedge @) \cdot \mathsf{res}(\mathsf{q}, (\mathsf{p(a)} \leftarrow \top) \cdot (\mathsf{p(b)} \leftarrow \top), \mathsf{p(c)} \wedge @) \cdot \bot \\
& \longrightarrow^* & (\mathsf{q} \wedge \mathsf{p(c)} \wedge @) \cdot \bot \\
& \equiv & \mathsf{q} \wedge \mathsf{p(c)} \wedge @
\end{array}$$

□

Bei der letzten unendlichen Ableitung zeigt sich, daß der Kalkül 3 zwar die prinzipielle Möglichkeit besitzt, sich „schlau“ zu verhalten; will man jedoch „schlaues“ Verhalten für eine Ableitung zusichern, muß man der Konstruktion bestimmte Kriterien zuordnen, die bei der Ableitung für ein gewisses Maß an „Fairness“ sorgen.

In dem obigen Beispiel kam die unendliche Ableitung dadurch zustande, daß wir **nie** das Literal q reduziert haben, wir waren also **unfair zu diesem Literal**.

Eine weiteres Phänomen wollen wir in dem folgenden Beispiel betrachten.

Beispiel 3.6 Sei P das Programm

$$(\mathsf{nat(0)} \leftarrow \top) \cdot (\mathsf{nat(s(X))} \leftarrow \mathsf{nat(X)})$$

und G die Anfrage

$$\mathsf{nat(X)}.$$

Was uns vorschwebt, ist, daß der Kalkül nach und nach ein System erzeugt, daß alle Lösungen für diese Anfrage enthält. Sind wir jedoch unfair, so können wir aber auch die folgende Ableitung herstellen:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{nat(X)} \wedge @\mathsf{(X)}} \\
\longrightarrow & \mathsf{res}(\mathsf{nat(X)}, (\mathsf{nat(s(Y))} \leftarrow \mathsf{nat(Y)}) \cdot (\mathsf{nat(0)} \leftarrow \top), @\mathsf{(X)}) \cdot \bot \\
\longrightarrow & (\mathsf{nat(Y)} \wedge @\mathsf{(s(Y))}) \cdot \mathsf{res}(\mathsf{nat(X)}, (\mathsf{nat(0)} \leftarrow \top), @\mathsf{(X)}) \cdot \bot \\
\longrightarrow^* & (\mathsf{nat(Z)} \wedge @\mathsf{(s(s(Z)))}) \cdot \mathsf{res}(\cdots) \cdot (\mathsf{nat(Y)} \wedge @\mathsf{(s(Y))}) \cdot \mathsf{res}(\cdots) \cdot \bot \\
\vdots &
\end{array}$$

Unsere „Unfairness“ besteht darin, daß wir uns einfach weigern, die Teile des Systems zu bearbeiten, die noch aus $\mathsf{res}(\cdots)$ bestehen. Dadurch verhindern wir insbesondere, daß wir die Lösung $\mathsf{0}$ für X erhalten. Wenn wir uns die obige Ableitung anschauen, stellen wir fest, daß es ausreichen würde, wenn wir nur jede Komponente des Systems, nicht jedoch jedes Literal fair behandeln. Diese Art Ableitungen zu konstruieren wäre also nur halbfair, oder „semifair“: Die nächste Ableitung ist so eine.

$$\mathsf{nat(X)} \wedge @\mathsf{(X)}$$

$$
\begin{array}{ll}
\longrightarrow & \mathsf{res}(\mathsf{nat}(\mathsf{X}), (\mathsf{nat}(0) \leftarrow \top) \cdot (\mathsf{nat}(\mathsf{s}(\mathsf{Y})) \leftarrow \mathsf{nat}(\mathsf{Y})), @(\mathsf{X})) \cdot \bot \\
\longrightarrow & @(0) \cdot \mathsf{res}(\mathsf{nat}(\mathsf{X}), \mathsf{nat}(\mathsf{s}(\mathsf{Y})) \leftarrow \mathsf{nat}(\mathsf{Y}), @(\mathsf{X})) \cdot \bot \\
\longrightarrow & @(0) \cdot (\mathsf{nat}(\mathsf{Y}) \wedge @(\mathsf{s}(\mathsf{Y}))) \cdot \bot \\
\longrightarrow^* & @(0) \cdot @(\mathsf{s}(0)) \cdot (\mathsf{nat}(\mathsf{Z}) \wedge @(\mathsf{s}(\mathsf{s}(\mathsf{Z})))) \cdot \bot \\
\vdots &
\end{array}
$$

Wenn wir jetzt unendlich lange „semifair“ reduzieren, bekommen wir also eine vollständige Antwortmenge. □

Jetzt wollen wir die in dem obigen Beispiel informell verwendeten Begriffe „fair“ und “semifair“ formalisieren. Die Idee dazu ist, Indices für Literale einzuführen, die sich bei jedem Reduktionsschritt erhöhen. Vorher müssen wir jedoch noch Begriffe festlegen, die beschreiben, welche Literale wir betrachten müssen.

Definition 3.7 *Ein Ziel G heißt* **Ziel eines Systems** S *genau dann, wenn:*

- *Es gibt ein System S' mit $S \equiv G \cdot S'$, oder*
- *Es gibt ein System S' mit $S \equiv \mathsf{res}(A, \cdots, H)$ und $G \equiv A \wedge H$.*

Ein Literal A heißt **Literal eines Systems** S *genau dann, wenn*

- *$A \neq @(\cdots)$, und*
- *Es gibt ein Ziel G des Systems S und ein Ziel H mit $G \equiv A \wedge H$.*

Definition 3.8 (Indizierte Ableitungen) *Wir ordnen jedem* **Literal eines Systems** *eine natürliche Zahl als Index zu. Wird ein Literal zu einem neuen Ziel reduziert, werden alle Literale der Klausel, die zur Reduktion herangezogen wurde, mit einem um eins erhöhten Index versehen.*
An Hand unseres Kalküls 3 bedeutet das die Erweiterung der folgenden Regel:

- $\mathsf{res}(A^i, (B \leftarrow B_1 \wedge \cdots \wedge B_n) \cdot Q, G) \cdot S \longrightarrow$
- $(\theta B_1^{i+1} \wedge \cdots \wedge \theta B_n^{i+1} \wedge \theta G) \cdot \mathsf{res}(A^i, Q, G) \cdot S$

falls θ prinzipaler Unifikator von $A \doteq B$.

Wenn wir eine Ableitung mit einem Ziel beginnen, versehen wir die Literale dieses Ziels alle mit dem Index 0.

Die oben definierte Indizierung von Literalen gibt uns jetzt die Kriterien für Semifairness und Fairness an die Hand.

Definition 3.9 (Ableitungen) *Sei*

$$A_1^0 \wedge \cdots \wedge A_n^0 \wedge @(\overline{x}) = S_0 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow \ldots$$

eine Ableitung im Kalkül 3.
Die Abbildung I ordnet einem Ziel G die Summe der Indices aller Literale des Zieles G zu.
Diese Ableitung heißt **semifair**, *falls*

- *sie endlich und nicht weiter fortsetzbar ist, oder*
- *unendlich ist, und es gilt:*
$$\lim_{i\to\infty} (\min \{I(G) \mid G \textit{ ist Ziel des Systems } S_i\}) = \infty$$

Sie heißt **fair**, *falls*

- *sie endlich und nicht weiter fortsetzbar ist, oder*
- *unendlich ist, und es gilt:*
$$\lim_{i\to\infty} \left(\min\left\{j \mid A^j \textit{ ist Literal des Systems } S_i\right\}\right) = \infty$$

Eine direkte Folgerung aus der obigen Definition ist:

Proposition 3.7 *Jede faire Ableitung ist semifair.*

Die Beispiele, die uns auf diese Begriffe gestoßen haben, betrachten wir jetzt noch einmal im Licht unserer formalen Definition.

Beispiel 3.7 (Fortsetzung Beispiel 3.5) Wir betrachten noch einmal einen Schritt in der unendlichen Ableitung aus Beispiel 3.5. Indiziert sieht diese so aus:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{q}^0 \wedge \mathsf{p(c)}^0 \wedge @} \\
\equiv & \left(\mathsf{q}^0 \wedge \mathsf{p(c)}^0 \wedge @\right) \cdot \bot \\
\longrightarrow & \mathsf{res}(\mathsf{q}^0, (\mathsf{p(a)} \leftarrow \top) \cdot (\mathsf{p(b)} \leftarrow \top) \cdot (\mathsf{q} \leftarrow \mathsf{q}), \mathsf{p(c)}^0 \wedge @) \cdot \bot \\
\equiv & \mathsf{res}(\mathsf{q}^0, (\mathsf{q} \leftarrow \mathsf{q}) \cdot (\mathsf{p(a)} \leftarrow \top) \cdot (\mathsf{p(b)} \leftarrow \top), \mathsf{p(c)}^0 \wedge @) \cdot \bot \\
\longrightarrow & \left(\mathsf{q}^1 \wedge \mathsf{p(c)}^0 \wedge @\right) \cdot \mathsf{res}(\mathsf{q}^0, (\mathsf{p(a)} \leftarrow \top) \cdot (\mathsf{p(b)} \leftarrow \top), \mathsf{p(c)}^0 \wedge @) \cdot \bot \\
\longrightarrow^* & \left(\mathsf{q}^1 \wedge \mathsf{p(c)}^1 \wedge @\right) \cdot \bot \\
\equiv & \mathsf{q}^1 \wedge \mathsf{p(c)}^0 \wedge @
\end{array}$$

Damit können wir diese Ableitung herstellen:

$$\begin{array}{rcl}
\mathsf{q}^0 \wedge \mathsf{p(c)}^0 \wedge @ & \longrightarrow^* & \mathsf{q}^1 \wedge \mathsf{p(c)}^0 \wedge @ \\
& \longrightarrow^* & \mathsf{q}^2 \wedge \mathsf{p(c)}^0 \wedge @ \\
& \longrightarrow^* & \mathsf{q}^3 \wedge \mathsf{p(c)}^0 \wedge @ \\
& \vdots &
\end{array}$$

Diese ist **semifair**, **nicht** jedoch **fair**. □

Beispiel 3.8 (Fortsetzung Beispiel 3.6) Die erste Ableitung aus Beispiel 3.6 sieht indiziert wie folgt aus:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{nat}(\mathsf{X})^0 \wedge @(\mathsf{X})} \\
\longrightarrow & \mathsf{res}(\mathsf{nat}(\mathsf{X})^0, (\mathsf{nat}(\mathsf{s}(\mathsf{Y})) \leftarrow \mathsf{nat}(\mathsf{Y})) \cdot (\mathsf{nat}(0) \leftarrow \top), @(\mathsf{X})) \cdot \bot \\
\longrightarrow & (\mathsf{nat}(\mathsf{Y})^1 \wedge @(\mathsf{s}(\mathsf{Y}))) \cdot \mathsf{res}(\mathsf{nat}(\mathsf{X})^0, (\mathsf{nat}(0) \leftarrow \top), @(\mathsf{X})) \cdot \bot \\
\longrightarrow^* & (\mathsf{nat}(\mathsf{Z})^2 \wedge @(\mathsf{s}(\mathsf{s}(\mathsf{Z})))) \cdot \mathsf{res}(\cdots) \cdot (\mathsf{nat}(\mathsf{Y})^1 \wedge @(\mathsf{s}(\mathsf{Y}))) \cdot \mathsf{res}(\cdots) \cdot \bot \\
\vdots &
\end{array}
$$

Leicht identifiziert man diese Ableitung als **nicht semifair**.

Die zweite Ableitung sieht folgendermassen aus:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{nat}(\mathsf{X})^0 \wedge @(\mathsf{X})} \\
\longrightarrow & \mathsf{res}(\mathsf{nat}(\mathsf{X})^0, (\mathsf{nat}(0) \leftarrow \top) \cdot (\mathsf{nat}(\mathsf{s}(\mathsf{Y})) \leftarrow \mathsf{nat}(\mathsf{Y})), @(\mathsf{X})) \cdot \bot \\
\longrightarrow & @(0) \cdot \mathsf{res}(\mathsf{nat}(\mathsf{X})^0, \mathsf{nat}(\mathsf{s}(\mathsf{Y})) \leftarrow \mathsf{nat}(\mathsf{Y}), @(\mathsf{X})) \cdot \bot \\
\longrightarrow & @(0) \cdot (\mathsf{nat}(\mathsf{Y})^1 \wedge @(\mathsf{s}(\mathsf{Y}))) \cdot \bot \\
\longrightarrow^* & @(0) \cdot @(\mathsf{s}(0)) \cdot (\mathsf{nat}(\mathsf{Z})^2 \wedge @(\mathsf{s}(\mathsf{s}(\mathsf{Z})))) \cdot \bot \\
\vdots &
\end{array}
$$

Diese Ableitung ist also **fair**. □

Satz 3.8 (Eigenschaften von Kalkül 3) *Seien P, G und $\overline{x}$ zulässig. Dann gelten die folgenden Aussagen:*

1. *Aus*
$$G \wedge @(\overline{x}) \longrightarrow^* @(\overline{s}) \cdot S$$
folgt, daß $[\overline{x}/\overline{s}] \in ANT_P(G)$.

2. *Aus*
$$G \wedge @(\overline{x}) \longrightarrow^* \bot$$
folgt, daß $\emptyset$ eine vollständige Antwortmenge für G bezüglich P.

3. *Wenn*
$$G \wedge @(\overline{x}) \longrightarrow^* @(\overline{s}_1) \cdot \cdots \cdot @(\overline{s}_n),$$
dann ist
$$\{[\overline{x}/\overline{s}_1], \ldots, [\overline{x}/\overline{s}_n]\}$$
eine vollständige Antwortmenge für G bezüglich P.

4. *Wenn*
$$G \wedge @(\overline{x}) = S_0 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow \cdots$$
eine semifaire Ableitung ist, dann ist
$$\bigcup_{i \geq 0} \{[\overline{x}/\overline{s}] \mid \textit{Es existiert } S: \ S_i \equiv @(\overline{s}) \cdot S\}$$
eine vollständige Antwortmenge für G bezüglich P.

Weiterhin gilt die folgende, sehr starke Aussage:

Satz 3.9 *Seien P, G und $\overline{x}$ zulässig, und $G \wedge @(\overline{x}) \longrightarrow^* \bot$. Dann gilt:*

Jede *faire Ableitung ist endlich und reduziert zu $\bot$.*

Bisher haben wir nur eine sehr schwache deklarative Aussage für den Fall kennengelernt, der in dem letzten Satz aufgegriffen wurde. Um stärkere Aussagen zu gewinnen, werden wir im nächsten Abschnitt etwas weiter ausholen.

3.6 Die Clarksche Vervollständigung

Bis jetzt haben wir deklarative Aussagen der Form $P \models \theta G$ betrachtet. Durch diese Charakterisierung haben wir beschrieben, wann ein θ eine Antwort ist. Diese Art reicht jedoch nicht aus, zu beschreiben, wann ein Ziel G **keine Antwort** hat. Ein Beispiel dazu ist das folgende.

Beispiel 3.9 (Fortsetzung Beispiel 3.6) Wir betrachten noch einmal das Programm aus Beispiel 3.6. Bezüglich dieses Programmes haben wir die Ableitung:

$$\mathsf{nat}(\mathsf{a}) \wedge @ \longrightarrow^* \bot.$$

Deklarativ haben wir jedoch:

$$P \not\models \mathsf{nat}(\mathsf{a}) \qquad \text{und} \qquad P \not\models \neg\mathsf{nat}(\mathsf{a}).$$

Einsehen läßt sich dieser Sachverhalt wie folgt: Für das kleinste Herbrand–Modell $\mathcal{H}(P)$ von P gilt gerade:

$$\mathcal{H}(P) \models \neg\mathsf{nat}(\mathsf{a}).$$

Für das Herbrand–Modell $\mathcal{A}$ von P, das der Interpretation $\mathcal{A}[\![\mathsf{rel}(\mathsf{nat}, 1)]\!]$ gerade noch den Wert a hinzufügt gilt:

$$\mathcal{A} \models \mathsf{nat}(\mathsf{a}).$$

$\square$

Unsere Intuition bei einem Programm sagt, daß da ja eigentlich keine Implikationen stehen sollten, sondern Äquivalenzen: Denn unsere Vorstellung sagt uns, daß $\mathsf{nat}(VX)$ **genau dann** gelten soll, wenn die Rümpfe der Klauseln gelten. Damit kommen wir zu der folgenden Transformation von Programmen in Formelmengen, die unserer Vorstellung eher entsprechen.

Definition 3.10 *Sei P ein Programm über der prädikatenlogischen Signatur Σ. Hieraus konstruieren wir die Formelmenge $P^{\odot}$ für $\odot \in \{\leftarrow, \leftrightarrow\}$ nach der folgenden Vorschrift.*

Für jedes Relationssymbol $\mathsf{rel}(a, n) \in \Sigma$ *wählen wir alle Klauseln*

$$\begin{array}{rcl} \mathsf{rel}(a,n)(\overline{t}_1) & \leftarrow & G_1 \\ & \vdots & \\ \mathsf{rel}(a,n)(\overline{t}_k) & \leftarrow & G_k \end{array}$$

aus P*, die dieses Relationssymbol im Kopf enthalten. Für diese Klauseln enthält* $P^{\odot}$ *die folgende Formel:*

$$\begin{array}{rcllcl} \mathsf{rel}(a,n)(\overline{x}) & \odot & \exists \overline{y}_1 (\overline{x} \doteq \overline{t}_1 & \wedge & G_1) \\ & \vee & \exists \overline{y}_2 (\overline{x} \doteq \overline{t}_2 & \wedge & G_2) \\ & & \quad \vdots & & \\ & \vee & \exists \overline{y}_k (\overline{x} \doteq \overline{t}_k & \wedge & G_k) \end{array}$$

Dabei ist $\overline{x}$ *eine Folge der Länge* n *paarweiser verschiedener, neuer*[2] *Variablen, und* $\mathcal{V}(\overline{y}_i) = \mathcal{V}(t_i) \cup \mathcal{V}(G_i)$.

Für jedes Relationssymbol $\mathsf{rel}(a, n)$*, daß nicht in dem Kopf einer Klausel des Programm auftritt, fügen wir* $P^{\odot}$ *die Klausel*

$$\mathsf{rel}(a,n)(\overline{x}) \odot \perp$$

hinzu.

Die folgende Proposition gibt die Intuition wieder, daß sich bei der Formelmenge $P^{\leftarrow}$, die wir im weiteren **homogene Form von** P nennen werden, gegenüber dem Ausgangsprogramm nichts wesentliches geändert hat.

Proposition 3.10 *Für alle Programme* P *gilt:*

$$P \text{ und } P^{\leftarrow} \text{ haben dieselben Modelle.}$$

Die zweite Formelmenge $P^{\leftrightarrow}$ wird eine für uns sehr wichtige Rolle spielen: Sie wird zusammen mit einer anderen Formelmenge eine in einem noch zu klärenden Sinne für uns angemessenere Semantik des Programmes P liefern. Bevor wir dieses jedoch erklären können, benötigen wir einige Begriffsbildungen.

Definition 3.11 (Theorie) *Sei eine bestimmte Signatur festgelegt.*

1. *Ein* **Satz** *ist eine Formel* φ *ohne freie Variablen, das heißt* $\mathcal{V}(\varphi) = \emptyset$.

2. *Eine Menge* Δ *von Sätzen heißt* **Theorie**.

3. *Eine Theorie* Δ *heißt* **vollständig***, falls für alle Sätze* φ *das folgende gilt:*

$$\Delta \models \varphi \qquad \text{oder} \qquad \Delta \models \neg\varphi.$$

[2]Neu heißt, daß diese Variablen nicht in den ursprünglichen Klauseln vorkamen.

4. *Eine Formelmenge Δ heißt* **konsistent**, *falls Δ ein Modell hat, das heißt, es gibt eine Struktur $\mathcal{A}$ mit $\mathcal{A} \models \Delta$.*

Für Theorien gelten die folgenden Aussagen.

Proposition 3.11

1. *Sei $\mathcal{A}$ eine Struktur. Dann ist*

$$\{\varphi \mid \varphi \textit{ Satz }, \mathcal{A} \models \varphi\}$$

   eine vollständige und konsistente Theorie.

2. *Seien $\mathcal{A}$ und $\mathcal{B}$ zwei Modelle einer vollständigen Theorie Δ. Dann gilt für alle Formeln φ:*

$$\mathcal{A} \models \varphi \iff \mathcal{B} \models \varphi.$$

3. *Sei Δ eine vollständige, rekursiv aufzählbare Theorie. Dann ist die Menge der aus Δ folgerbaren Sätze entscheidbar, das heißt:*

$$\{\varphi \mid \varphi \textit{ Satz}, \Delta \models \varphi\} \textit{ ist entscheidbar.}$$

4. *Sei $\mathcal{N}$ die Struktur der natürlichen Zahlen. Dann ist die Menge*

$$\{\varphi \mid \varphi \textit{ Satz}, \mathcal{N} \models \varphi\}$$

   nicht *rekursiv aufzählbar*[3].

Für uns ist die folgende Theorie wichtig, sie beschreibt als (unendliche) Menge von Formeln

Definition 3.12 (Die Theorie CET) *Sei Σ eine prädikatenlogische Signatur, die unendliche viele Funktionszeichen und mindestens eine Konstante enthält. Die Theorie* CET[4] *ist die Menge von Sätzen, die sich nach den folgenden Schemata bilden lassen:*

$$\begin{array}{llllllll}
\mathrm{Ax1}: & \forall(\, f(\overline{s}) & \doteq & f(\overline{t}) & \rightarrow & \overline{s} \doteq t\,) & & (f \in \Sigma) \\
\mathrm{Ax2}: & \forall(\, f(\overline{s}) & \doteq & g(\overline{t}) & \rightarrow & \perp\,) & & (f, g \in \Sigma, f \neq g) \\
\mathrm{Ax3}: & \forall(\quad x & \doteq & f(\overline{s}) & \rightarrow & \perp\,) & & (f \in \Sigma, x \in \mathcal{V}(\overline{s}))
\end{array}$$

Die angenehmen Eigenschaften der Theorie CET gibt der nächste Satz wieder:

Satz 3.12 *Die Theorie* CET *ist vollständig.*

[3] Diese klassische Aussage stammt von Kurt Gödel.
[4] CET steht für Clark's Equality Theory.

Definition 3.13 (Clarksche Vervollständigung) *Sei P ein Programm. Die* **Clarksche Vervollständigung von** P — $\text{comp}(P)$ [5] — *ist die folgende Formelmenge:*

$$\text{comp}(P) := \text{CET} \cup P^{\leftrightarrow}.$$

Die nächste Aussage gibt wieder, daß die Vervollständigung eines Programms zumindest nicht schwächer als das Programm selber ist.

Proposition 3.13 *Für alle Programme P gilt:*

$$\text{comp}(P) \models P.$$

Für variablenfreie Ziele ändert sich die Folgerbarkeit bezüglich P und $\text{comp}(P)$ nicht, dies sagt der nächste Satz.

Satz 3.14 *Sei P ein Programm und G ein variablenfreies Ziel. Dann gilt:*

$$P \models G \iff \text{comp}(P) \models G$$

3.7 Nichtableitbarkeit im Kalkül 3

Jetzt sind wir endlich in der Lage, eine befriedigende deklarative Semantik des endlichen Scheiterns einer Ableitung im Kalkül 3 zu geben.

Satz 3.15 *Seien P, G und $\overline{x}$ zulässig. Dann gilt:*

$$G \wedge @(\overline{x}) \longrightarrow^{*} \bot \iff \text{comp}(P) \models \neg\exists\overline{x}\, G$$

.

Man bemerke, daß dieser Satz gerade unsere Intuition, die wir an dem Beispiel 3.9 gewonnen haben, widerspiegelt. Das nächste Beispiel wird jedoch zeigen, daß die Vervollständigung eines Programmes in der Regel keine vollständige Theorie ist.

Beispiel 3.10 (Graphen) Sei der Graph aus Abbildung 3.4 gegeben. Wir wollen ein Programm angeben, daß ein einstelliges Prädikat $\mathsf{reach(X)}$ enthält, das gelten soll, wenn X von dem Knoten a erreichbar ist. Das Programm P sieht wie folgt aus

$$\begin{array}{lll}
\mathsf{arc(a,b)}^{6} & & \\
\mathsf{arc(c,d)} & & \\
\mathsf{arc(d,c)} & & \\
\mathsf{reach(a)} & & \\
\mathsf{reach(X)} & \leftarrow & \mathsf{reach(Y)} \wedge \mathsf{arc(Y,X)}
\end{array}$$

[5] comp für „completion“, engl. Vervollständigung

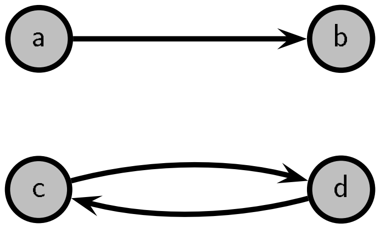


Abbildung 3.4: Ein Graph

Leicht ist zu sehen, daß der Knoten c nicht von a aus erreichbar ist, das heißt:

$$P \not\models \mathsf{reach(c)}.$$

Anders ausgedrückt haben wir also:

$$ANT_P(\mathsf{reach(c)}) = \emptyset.$$

Wenn wir jetzt die Anfrage $\mathsf{reach(c)}$ stellen, und wir eine faire Ableitung konstruieren, möchten wir, daß wir zu $\perp$ reduzieren. Also betrachten wir eine Ableitung:

$$\begin{array}{rll}
\mathsf{r(c)}^0 \wedge @ & \longrightarrow^* & \mathsf{reach(X)}^1 \wedge \mathsf{arc(X,c)}^1 \wedge @ \\
 & \longrightarrow^* & \mathsf{reach(d)}^2 \wedge @ \\
 & \longrightarrow^* & \mathsf{reach(Y)}^3 \wedge \mathsf{arc(Y,d)}^3 \wedge @ \\
 & \longrightarrow^* & \mathsf{reach(c)}^4 \wedge @ \\
 & \longrightarrow^* & \mathsf{reach(Z)}^5 \wedge \mathsf{arc(Z,c)}^5 \wedge @ \\
 & \vdots &
\end{array}$$

Wir sehen also, daß (nach Satz 3.9) es keine Ableitung gibt, so daß die Anfrage $\mathsf{reach(c)} \wedge @$ zu $\perp$ reduzieren kann. Wie verhält sich dann comp(P)?

Die Vervollständigung von P enthält dann die folgenden Formeln (wir geben nur die Teile von CET an, deren Funktionszeichen in P auftreten):

$$\left.\begin{array}{rcl}
\mathsf{arc(X,Y)} & \leftrightarrow & \mathsf{X} \doteq \mathsf{a} \wedge \mathsf{Y} \doteq \mathsf{b} \\
 & \vee & \mathsf{X} \doteq \mathsf{c} \wedge \mathsf{Y} \doteq \mathsf{d} \\
 & \vee & \mathsf{X} \doteq \mathsf{d} \wedge \mathsf{Y} \doteq \mathsf{c} \\
\mathsf{r(Z)} & \leftrightarrow & \mathsf{Z} \doteq \mathsf{a} \\
 & \vee & \exists \mathsf{Y} \exists \mathsf{X}\, (\mathsf{Z} \doteq \mathsf{X} \wedge \mathsf{r(Y)} \wedge \mathsf{arc(Y,X)})
\end{array}\right\} P^{\leftrightarrow}$$

$$\left.\begin{array}{lll}
\mathsf{a} \doteq \mathsf{b} \;\rightarrow\; \perp & \mathsf{a} \doteq \mathsf{c} \;\rightarrow\; \perp & \mathsf{a} \doteq \mathsf{d} \;\rightarrow\; \perp \\
\mathsf{b} \doteq \mathsf{a} \;\rightarrow\; \perp & \mathsf{b} \doteq \mathsf{c} \;\rightarrow\; \perp & \mathsf{b} \doteq \mathsf{d} \;\rightarrow\; \perp \\
\mathsf{c} \doteq \mathsf{a} \;\rightarrow\; \perp & \mathsf{c} \doteq \mathsf{b} \;\rightarrow\; \perp & \mathsf{c} \doteq \mathsf{d} \;\rightarrow\; \perp \\
\mathsf{d} \doteq \mathsf{a} \;\rightarrow\; \perp & \mathsf{d} \doteq \mathsf{b} \;\rightarrow\; \perp & \mathsf{d} \doteq \mathsf{c} \;\rightarrow\; \perp \\
\vdots & \vdots & \vdots
\end{array}\right\} \text{CET}$$

[6]Hier und im weiteren lassen wir der Kürze wegen das Ziel $\top$ weg.

Aber auch die Vervollständigung sagt das folgende aus:

$$\mathrm{comp}(P) \not\models \mathsf{reach(c)} \qquad \text{und} \qquad \mathrm{comp}(P) \not\models \neg\mathsf{reach(c)}.$$

Das zeigen wir durch Konstruktion zweier Herbrand–Modelle $\mathcal{H}_1$ und $\mathcal{H}_2$ von comp(P), für die gilt:

$$\mathcal{H}_1 \models \neg\mathsf{reach(c)} \qquad \text{und} \qquad \mathcal{H}_2 \models \mathsf{reach(c)}.$$

Die Konstruktion lautet:

$$\mathcal{H}_1 \ : \ \begin{cases} \mathcal{H}_1 [\![\mathsf{rel(arc,2)}]\!] & := \ \{(\mathsf{a},\mathsf{b}),(\mathsf{c},\mathsf{d}),(\mathsf{d},\mathsf{c})\} \\ \mathcal{H}_1 [\![\mathsf{rel(reach,1)}]\!] & := \ \{(\mathsf{a}),(\mathsf{b})\} \end{cases}$$

$$\mathcal{H}_2 \ : \ \begin{cases} \mathcal{H}_2 [\![\mathsf{rel(arc,2)}]\!] & := \ \{(\mathsf{a},\mathsf{b}),(\mathsf{c},\mathsf{d}),(\mathsf{d},\mathsf{c})\} \\ \mathcal{H}_2 [\![\mathsf{rel(reach,1)}]\!] & := \ \{(\mathsf{a}),(\mathsf{b}),(\mathsf{c}),(\mathsf{d})\} \end{cases}$$

Das Modell $\mathcal{H}_1$ — dies ist gerade das kleinste Herbrand–Modell — entspricht der intendierten Bedeutung, das zweite Modell ist gerade so vergrößert, daß es die zweite Klausel für reach erfüllt.

Wir sehen also insbesondere, daß comp(P) **keine vollständige** Theorie ist! □

3.8 Der Kalkül 4

In diesem Abschnitt streben wir an, neben Literalen auch negierte Literale in einem Kalkül ableiten zu können. Negation läßt sich oft gewinnbringend einsetzen. Betrachten wir dazu ein Beispiel.

Beispiel 3.11 (Negation in Programmen) Gegeben seien zwei Listen, L und L′. Gesucht ist ein Prädikat diff(X, L, L′), das genau dann gilt, wenn X in der Differenz der Listen L und L′ ist. Die Umsetzung ist trivial, das folgende Programm leistet das Gewünschte, vorausgesetzt, wir dürfen die logische Negation verwenden.

$$\begin{array}{rcl} \mathsf{diff(X,L,L')} & \leftarrow & \mathsf{member(X,L)} \\ & \wedge & \neg\mathsf{member(X,L')} \end{array}$$

$$\begin{array}{l} \mathsf{member(X,cons(X,L))} \\ \mathsf{member(X,cons(Y,L))} \ \leftarrow \ \mathsf{member(X,L)} \end{array}$$

□

In Kalkül 3 können wir neben der erfolgreichen Berechnung von Antworten auch feststellen, wenn es keine Antworten gibt, das heißt, wenn ein System zu $\bot$ reduziert. Damit haben wir schon eine intuitive Vorstellung, wie wir die Negation eines Literals operational in unserem Kalkül behandeln können. Intuitiv haben wir:

$$\frac{A \longrightarrow^* \bot}{\neg A \longrightarrow \top} \qquad\qquad \frac{A \longrightarrow^* \top}{\neg A \longrightarrow \bot}.$$

Diese Idee werden wir jetzt im Kalkül 4 aufgreifen, der in Abbildung 3.5 definiert ist. Der Kalkül 4 unterscheidet sich in einigen Punkten von Kalkül 3, diese werden wir jetzt diskutieren:

Syntax

	A, B	$::=$	$a(t_1, \ldots, t_n)$
Literale:	L	$::=$	$A \mid \neg A$
Ziele:	G, H	$::=$	$L \mid \top \mid G \wedge H$
Klauseln:			$A \leftarrow G$
Konfigurationen:			$\mathsf{res}(A, P, G)$
Systeme:	S	$::=$	$G \mid \bot \mid S \cdot S$
Programme:	P, Q	$::=$	$A \leftarrow G \mid \bot \mid P \cdot Q$

Kongruenz

- Siehe Abbildung 3.3.

Reduktion

- $$\frac{\mathsf{res}(A, Q, G) \longrightarrow S}{(A \wedge G) \cdot S' \longrightarrow S \cdot S'}$$

  falls:

  - Q Variante von P;
  - Q variablendisjunkt zu $A \wedge G$;
  - $A \neq @(\cdots)$.

- $$\frac{A \longrightarrow^* \bot}{(\neg A \wedge G) \cdot S \longrightarrow G \cdot S}$$

  falls A variablenfrei.

- $$\frac{A \longrightarrow^* \top \cdot S'}{(\neg A \wedge G) \cdot S \longrightarrow S}$$

  falls A variablenfrei.

- $$\frac{\mathsf{res}(A, Q, G) \longrightarrow S \qquad \mathsf{res}(A, Q', G) \longrightarrow S'}{\mathsf{res}(A, Q \cdot Q', G) \longrightarrow S \cdot S'}$$

  falls $Q \neq \bot$ und $Q' \neq \bot$.

- $\mathsf{res}(A, B \leftarrow H, G) \longrightarrow \bot$

  falls A und B nicht unifizierbar.

- $\mathsf{res}(A, B \leftarrow H, G) \longrightarrow \theta H \wedge \theta G$

  falls θ prinzipaler Unifikator von $A \doteq B$.

Abbildung 3.5: Der Kalkül 4

- Als Literale sind jetzt auch negierte atomare Formeln zugelassen, das ist gerade die echte Erweiterung in der Syntax von Kalkül 3 zu Kalkül 4.

- Eine weitere Änderung in der Syntax betrifft die Einführung von **Konfigurationen**. Diese beschreiben zusammen mit den abgewandelten Ableitungsregeln für Konfigurationen eine Entwurfsalternative, die auch in Kalkül 3 möglich gewesen wäre. In Kalkül 3 war es erlaubt, die Reduktion von Konfigurationen auf gleicher Ebene mit der Reduktion anderer Literale voranzutreiben. Hier besteht jedoch eine Partitionierung der Reduktion: Zuerst muß die Reduktion einer Konfiguration zu einem System ganz abgeschlossen werden, erst dann kann mit der Reduktion des Systems weiter fortgefahren werden. Dieses kommt der Intuition entgegen, daß wir nur an dem System interessiert sind, daß sich aus einem gegebenen Ziel und dem Programm bilden läßt, nicht jedoch an seiner Herleitung. Auch der Fall eines leeren Programmes hat sich geändert: In diesem Trivialfall kann eine Konfiguration **nicht** reduziert werden.

- Die Behandlung eines negativen Literals[7] erfordert auch den rekursiven Aufbau der Ableitungsrelation.

In direkter Art und Weise werden wir auch in Kalkül 4 die syntaktischen Objekte des Kalküls mit Formeln und Formelmengen identifizieren. Die einzige Änderung besteht darin, jetzt auch überall negierte Literale zuzulassen. Insbesondere verwenden wir jetzt auch wieder den Begriff der Vervollständigung eines Programms, wo wir implizit die Erweiterung um negierte Literale einschließen.

Die Idee hinter Kalkül 4 ist genau die weiter oben eingeführte: Wende „$\longrightarrow$" rekursiv an, und genau dann, wenn A zu $\perp$ reduziert, reduziert $\neg A$ zu $\top$. Diese Art der Behandlung der Negation als Nichtableitbarkeit wird **Negation als Scheitern** oder **Negation als endliches Scheitern** genannt[8].

Eine zusätzliche Einschränkung in der obigen Definition ist, daß wir verlangen, daß A **geschlossen** ist, das heißt keine freien Variablen enthält. Das folgende Beispiel wird uns den Grund dafür liefern.

Beispiel 3.12 (Inkonsistenz) Sei P das Programm

$$\begin{gathered}\mathsf{p(a)}\\ \mathsf{q(b)}\end{gathered}$$

und G die Anfrage

$$\mathsf{p(X)} \wedge \neg\mathsf{q(X)} \wedge @\mathsf{(X)}.$$

Wenn wir in der Definition von Kalkül 4 die Bedingung der Variablenfreiheit außer acht lassen, können wir die folgende Ableitung herstellen.

$$\mathsf{p(X)} \wedge \neg\mathsf{q(X)} \wedge @\mathsf{(X)}$$

[7] Wir wollen ein Literal L **negativ** nennen, wenn es die Form $\neg A$ hat; ansonsten **positiv**.

[8] Die englischen Termini in der Literatur sind **negation as failure**, resp. **negation as finite failure**.

$$
\begin{array}{ll}
\equiv & \neg q(X) \wedge p(X) \wedge @(X) \\
& \boxed{\begin{array}{ll} q(X) & \\ & \boxed{res(q(X), p(a) \cdot q(b), \top) \quad \longrightarrow^{*} \quad \top} \\ \longrightarrow & \top \cdot \bot \end{array}}^{9} \\
\longrightarrow & \bot
\end{array}
$$

Eine korrekte Ableitung in Kalkül 4 liefert uns:

$$
\begin{array}{ll}
\multicolumn{2}{l}{p(X) \wedge \neg q(X) \wedge @(X)} \\
& \boxed{res(p(X), p(a) \cdot q(b), \neg q(X) \wedge @(X)) \quad \longrightarrow^{*} \quad \neg q(a) \wedge @(a)} \\
\longrightarrow & \neg q(a) \wedge @(a) \\
& \boxed{\begin{array}{ll} q(a) & \\ & \boxed{res(q(a), p(a) \cdot q(b), \top) \quad \longrightarrow^{*} \quad \bot} \\ \longrightarrow & \bot \end{array}} \\
\longrightarrow & @(a)
\end{array}
$$

Das heißt, daß wir bei Nichtbeachtung der Variablenfreiheit zwei **essentiell** verschiedene Ableitungen:

$$
p(X) \wedge \neg q(X) \wedge @(X) \quad
\begin{array}{ll}
\nearrow^{*} & \bot \\
& \\
\searrow_{*} & @(a)
\end{array}
$$

herstellen können. Durch Nichtbeachtung der Variablenfreiheit würden wir somit die **Konsistenz** des Kalküls zerstören. □

Auch die Programmiersprache Prolog folgt dieser Art der Negation, sie läßt jedoch die Bedingung der Variablenfreiheit aus, um die Abarbeitung zu vereinfachen.

Nun wollen wir uns der deklarativen Semantik dieser Negation zuwenden. Dazu betrachten wir zuerst ein Beispiel.

Beispiel 3.13 Sei P das Programm

$$
\begin{array}{c}
arc(a, b) \\
arc(d, c)
\end{array}
$$

Die Anfrage $\neg arc(b, a) \wedge @$ reduziert zu $@$, aber es gilt:

$$P \not\models \neg arc(b, a).$$

[9] Die Ableitungen in den eingerahmten Kästen geben gerade die rekursiven Schritte in der Definition der Ableitbarkeit wieder. Um die Übersichtlichkeit zu erhöhen, unterschlagen wir die schrittweise Reduktion von Konfigurationen.

Die obige Semantik betrachtet alle Modelle des Programmes P. Dies sind jedoch generell zu viele. Wenn wir jedoch zur Vervollständigung comp(P) von P übergehen, haben wir

$$\mathrm{comp}(P) \models \neg \mathsf{arc}(\mathsf{b}, \mathsf{a}).$$

Intuitiv liegt das gerade daran, daß die Vervollständigung

$$\left.\begin{array}{rcl} \mathsf{arc}(\mathsf{X}, \mathsf{Y}) & \leftrightarrow & \mathsf{X} \doteq \mathsf{a} \wedge \mathsf{Y} \doteq \mathsf{b} \\ & \vee & \mathsf{X} \doteq \mathsf{d} \wedge \mathsf{Y} \doteq \mathsf{c} \end{array}\right\} P^{\leftrightarrow}$$

$$\left.\begin{array}{ccccccccc} \mathsf{a} \doteq \mathsf{b} & \rightarrow & \perp & \mathsf{a} \doteq \mathsf{c} & \rightarrow & \perp & \mathsf{a} \doteq \mathsf{d} & \rightarrow & \perp \\ \mathsf{b} \doteq \mathsf{a} & \rightarrow & \perp & \mathsf{b} \doteq \mathsf{c} & \rightarrow & \perp & \mathsf{b} \doteq \mathsf{d} & \rightarrow & \perp \\ \mathsf{c} \doteq \mathsf{a} & \rightarrow & \perp & \mathsf{c} \doteq \mathsf{b} & \rightarrow & \perp & \mathsf{c} \doteq \mathsf{d} & \rightarrow & \perp \\ \mathsf{d} \doteq \mathsf{a} & \rightarrow & \perp & \mathsf{d} \doteq \mathsf{b} & \rightarrow & \perp & \mathsf{d} \doteq \mathsf{c} & \rightarrow & \perp \\ & \vdots & & & \vdots & & & \vdots & \end{array}\right\} \mathrm{CET}$$

genau beschreibt, wie die beiden Argumente von arc auszusehen haben, und CET zusätzlich ausdrückt, daß alle Konstanten wirklich verschieden sind. □

Damit kommen wir zur Formulierung einer Korrektheitsaussage für Kalkül 4.

Definition 3.14 (Korrektheit von Kalkül 4) *Seien P, G und $\overline{x}$ zulässig. Dann gelten die beiden folgenden Aussagen:*

1. *Wenn*
$$G \wedge @(\overline{x}) \longrightarrow^* \perp,$$
dann
$$\mathrm{comp}(P) \models \neg \exists \overline{x}\, G.$$

2. *Wenn*
$$G \wedge @(\overline{x}) \longrightarrow^* @(\overline{s}) \cdot S,$$
dann
$$\mathrm{comp}(P) \models [\overline{x}/\overline{s}]G.$$

Durch die Hinzunahme von negierten Literalen verliert man zum Beispiel auch die angenehme Eigenschaft, daß eine Formelmenge ein kleinstes Herbrand-Modell besitzt. Betrachten wir dazu das nächste Beispiel.

Beispiel 3.14 (Kein eindeutiges minimales $\mathcal{H}$-Modell) Sei P das Programm

$$\begin{array}{rcl} \mathsf{p} & \leftarrow & \neg \mathsf{q} \\ \mathsf{q} & \leftarrow & \neg \mathsf{p} \end{array}$$

Beide Programmklauseln sind logisch äquivalent zu $\mathsf{p} \vee \mathsf{q}$, das heißt:

$$\mathsf{p} \leftarrow \neg \mathsf{q} \quad \models\!\!\!\mid \quad \mathsf{q} \leftarrow \neg \mathsf{p} \quad \models\!\!\!\mid \quad \mathsf{p} \vee \mathsf{q}.$$

Damit gibt es **zwei minimale** Herbrand-Modell $\mathcal{H}_1$ und $\mathcal{H}_2$ von P:

$$\begin{array}{lcl} \mathcal{H}_1 & : & \{\mathsf{rel}(\mathsf{p}, 0)\} \\ \mathcal{H}_2 & : & \{\mathsf{rel}(\mathsf{q}, 0)\} \end{array}$$

Anders ausgedrückt: Es gibt kein kleinstes Herbrand-Modell von P.

Wenn wir zu comp(P) übergehen, erhalten wir als $P^{\leftrightarrow}$ die Formelmenge:

$$\begin{array}{lcl} \mathsf{p} & \leftrightarrow & \neg\mathsf{q} \\ \mathsf{q} & \leftrightarrow & \neg\mathsf{p} \end{array}$$

Auch diese beiden Formeln sind wieder zueinander logisch äquivalent, und es gilt:

$$\mathsf{p} \leftrightarrow \neg\mathsf{q} \quad \models\!\!\!\!\!\!\;\dashv \quad \mathsf{q} \leftrightarrow \neg\mathsf{p} \quad \models\!\!\!\!\!\!\;\dashv \quad (\neg\mathsf{p} \land \mathsf{q}) \lor (\mathsf{p} \land \neg\mathsf{q}).$$

Damit sieht man, daß $\mathcal{H}_1$ und $\mathcal{H}_2$ wiederum **zwei minimale** Modelle von comp(P) sind. $\square$

Eine Vollständigkeitsaussage für den Kalkül 4 werden wir nicht angeben. Das Phänomen der Vollständigkeit eines Kalküles mit Negation werden wir später noch intensiv untersuchen. Wir wollen zumindest herausarbeiten, daß eine Vollständigkeitsaussage der Form

$$\mathrm{comp}(P) \models \exists \overline{x}\, G \quad \Longrightarrow \quad G \land @(\overline{x}) \longrightarrow^{*} @(\cdots) \cdot S$$

nicht möglich ist.

Beispiel 3.15 (Inkonsistenz von comp(P)**)** Daß die Vollständigkeit nicht gilt, kann man leicht an einem Beispiel einsehen, bei dem comp(P) inkonsistent ist. Für das Programm P

$$\mathsf{p} \leftarrow \neg\mathsf{p}$$

ist $P^{\leftrightarrow}$ gleich

$$\mathsf{p} \leftrightarrow \neg\mathsf{p}$$

Das heißt comp(P) ist inkonsistent, und die Vollständigkeit ist auf triviale Art und Weise zerstört. $\square$

Ein zweites Beispiel ist das folgende.

Beispiel 3.16 („Zappelnde“ Ableitungen) Sei P das Programm

$$\begin{array}{lcl} \mathsf{r(a)} & & \\ \mathsf{r(b)} & & \\ \mathsf{q} & \leftarrow & \neg\mathsf{r(X)} \end{array}$$

Damit berechnet sich $P^{\leftrightarrow}$ zu

$$\begin{array}{lcl} \mathsf{r(X)} & \leftrightarrow & \mathsf{X} \doteq \mathsf{a} \\ & \lor & \mathsf{X} \doteq \mathsf{b} \\ \mathsf{q} & \leftrightarrow & \exists \mathsf{X}\, \neg\mathsf{r(X)} \end{array}$$

Da wir bei der Definition von CET vorausgesetzt hatten, daß es unendlich viele Funktionszeichen gibt, gibt es zumindest eins, z.B. f. Es gilt also

$$\mathrm{comp}(P) \models \neg\mathsf{r}(\mathsf{f}(\mathsf{a})),$$

und damit auch

$$\mathrm{comp}(P) \models \exists\mathsf{X}\ \neg\mathsf{r}(\mathsf{X}).$$

Damit liefert die Äquivalenz für q:

$$\mathrm{comp}(P) \models \mathsf{q}.$$

Um Vollständigkeit zu erhalten, muß also auch $\mathsf{q} \wedge @$ zu $@$ reduzieren. Betrachten wir also die folgende Ableitung:

$$\begin{array}{ll} \mathsf{q} \wedge @ & \\ & \boxed{\mathsf{res}(\mathsf{q}, (\mathsf{q} \leftarrow \neg\mathsf{r}(\mathsf{X})) \cdot \mathsf{r}(\mathsf{a}) \cdot \mathsf{r}(\mathsf{b}), @) \quad \longrightarrow^* \quad \neg\mathsf{r}(\mathsf{X}) \wedge @} \\ \longrightarrow & \neg\mathsf{r}(\mathsf{X}) \wedge @ \end{array}$$

Diese Ableitung läßt sich **nicht** mehr weiter fortsetzen[10], da das negierte Literal noch die freie Variable X enthält. Also ist auch in diesem Beispiel die Vollständigkeit nicht gegeben. □

[10] In der englischen Literatur werden Ableitungen, die sich aus diesem Grunde nicht mehr fortsetzen lassen, als **floundering** — to flounder heißt zappeln — bezeichnet.

Teil II

Logische Datenstrukturen

Die vielleicht wichtigste Innovation logischer Programmiersprachen ist die Tatsache, daß Programme mit Variablen rechnen können, deren Werte nur unvollständig und implizit beschrieben sind[1]. Zur Beschreibung der Variablen werden dabei prädikatenlogische Formeln verwendet, mit denen in sinnvoller Weise gerechnet werden kann.

Für Variablen, deren Werte Zahlen sind, ist uns das Rechnen mit logischen Beschreibungen aus der Schule bestens vertraut. Zum Beispiel haben wir gelernt, die implizite Beschreibung

$$x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 5 \wedge x - y = 1$$

in die explizite Beschreibung

$$x = 3 \wedge y = 2$$

umzurechnen. Obwohl die beiden Beschreibungen völlig unterschiedlich aussehen, sind sie doch äquivalent im logischen Sinne, das heißt, sie drücken beide dieselbe Information über die Werte der Variablen x und y aus.

Im folgenden werden wir uns allerdings weniger mit Zahlen[2], sondern hauptsächlich mit Bäumen beschäftigen. Dabei steht das maschinelle Rechnen mit den entsprechenden logischen Beschreibungen im Vordergrund. Um zyklische Datenstrukturen modellieren zu können, werden wir auch unendliche Bäume betrachten.

Außerdem werden wir uns mit sogenannten Merkmalsbäumen beschäftigen, die den aus imperativen Programmiersprachen bekannten Records oder Verbunden entsprechen. Wir werden Beschreibungen unterschiedlicher Expressivität betrachten, zum Beispiel solche mit Negation, Disjunktion und Quantifizierung.

Logische Beschreibungen mit denen die Maschine sinnvoll rechnen kann, werden wir im folgenden als Constraints bezeichnen. Das Rechnen mit Constraints bezeichnen wir oft als Simplifikation.

Das Wort Simplifikation soll zum Ausdruck bringen, daß es beim Rechnen mit Constraints vor allem darum geht, die enthaltene Information über die Werte der Variablen so explizit wie möglich darzustellen.

[1] In der Literatur wird diese Eigenschaft oft unter dem Begriff „logic variables“ subsumiert.

[2] Zahlen sind aber selbstverständlich auch bei logischen Programmiersprachen von großer Bedeutung.

Kapitel 4

Constraintsysteme

Ein **Constraintsystem** legt die Syntax und Semantik einer Menge von Constraints fest. Wir werden Constraintsysteme unter Rückgriff auf die Prädikatenlogik definieren. Dabei verwenden wir Prädikatenlogik mit Gleichheit, d. h. $\doteq$ ist ein logisches Symbol, das in jeder Struktur als Identität interpretiert werden muß. Wie üblich legen wir eine abzählbar unendliche Menge von Variablen fest, die wir mit x, y, z bezeichnen.

Definition 4.1 (Constraintsystem) *Ein Constraintsystem ist ein Tripel* $(\Sigma, \Delta, \mathrm{CON})$ *wie folgt:*

- Σ *ist eine prädikatenlogische Signatur.*
- Δ *ist eine konsistente Theorie über* Σ*.*
- CON *ist eine Menge von Formeln über* Σ*, so daß*
  - CON *enthält* $\bot$ *und* $\top$*;*
  - CON *ist abgeschlossen unter Konjunktion (das heißt wenn* $\varphi, \psi \in \mathrm{CON}$*, dann* $\varphi \wedge \psi \in \mathrm{CON}$*);*
  - CON *ist abgeschlossen unter Varianten (das heißt wenn* $\varphi \in \mathrm{CON}$ *und* ψ *Variante von* φ*, dann ist* $\psi \in \mathrm{CON}$*).*

Sei $\Gamma = (\Sigma, \Delta, \mathrm{CON})$ ein Constraintsystem. Eine Struktur $\mathcal{A}$ für Σ heißt **Modell von** Γ genau dann, wenn $\mathcal{A}$ ein Modell der Theorie Δ ist. Seien $\varphi, \psi \in \mathrm{CON}$. Wir sagen:

- φ **subsumiert** ψ **in** Γ ($\varphi \models_\Gamma \psi$), falls $\varphi \models_\Delta \psi$ (das heißt für jedes Modell $\mathcal{A}$ von Γ ist jede Lösung von φ auch eine Lösung von ψ, also $\mathcal{A}[\![\varphi]\!] \subseteq \mathcal{A}[\![\psi]\!]$).
- φ **dissubsumiert** ψ **in** Γ ($\varphi \models_\Gamma \neg\psi$), falls $\varphi \models_\Delta \neg\psi$.
- φ **ist erfüllbar in** Γ, falls $\varphi \not\models_\Gamma \bot$.
- φ **ist äquivalent zu** ψ **in** Γ ($\varphi \mathrel{|\!\!=\!\!|}_\Gamma \psi$), falls $\varphi \models_\Gamma \psi$ und $\psi \models_\Gamma \varphi$.

Dissubsumption und Erfüllbarkeit sind verschiedene Sprechweisen für dieselbe Eigenschaft, da

$$\varphi \models_\Gamma \neg\psi \iff \varphi \wedge \psi \models_\Gamma \bot.$$

Typischerweise ist die Menge der Constraints gegenüber der Menge aller Formeln über der Constraintsignatur stark eingeschränkt, um die gewünschten Berechnungsdienste mit angemessenem Aufwand erzielen können.

Man beachte, daß Erfüllbarkeit und Äquivalenz in Γ mit Hilfe der Subsumptionsrelation $\models_\Gamma$ ausgedrückt werden können. Andererseits kann die Subsumption mit Hilfe der Äquivalenz ausgedrückt werden, da

$$\varphi \models_\Gamma \psi \iff \varphi \models\!\mid_\Gamma \varphi \wedge \psi.$$

Beispiel 4.1 (Das Constraintsystem INT) Das Constraintsystem INT stellt Constraints für ganze Zahlen zur Verfügung. Die Signatur von INT enthält die Konstanten 0, -1, 1, -2, 2, $\ldots$, die zweistelligen Funktionssymbole $+$ und $*$, und das zweistellige Relationssymbol $\dot{<}$. Als Theorie von INT wählen wir alle Sätze, die in der Struktur der ganzen Zahlen gültig sind. Als Constraints von INT wählen wir

$$\varphi, \psi \quad ::= \quad \bot \quad | \quad \top \quad | \quad s \doteq t \quad | \quad s \mathrel{\dot{<}} t \quad | \quad \varphi \wedge \psi$$

wobei s und t beliebige Terme über der Signatur von INT sind. □

Die Constraints eines Constraintsystems stellen partielle Beschreibungen der Werte dar, die die Variablen in den Modellen des Systems annehmen können. Wir können auch sagen, daß ein Constraint Information über die Werte der Variablen enthält. Für jedes Modell stellt ein Constraint eine Relation zwischen seinen freien Variablen dar. Beispielsweise legt $\mathsf{X} + \mathsf{Y} \doteq \mathsf{5}$ im Standardmodell von INT eine Relation fest, die die Werte von X und Y bijektiv verknüpft. Konjunktion von Constraints kann als informationsverknüpfende Operation gesehen werden.

Die Subsumptionsrelation ist eine Quasi-Ordnung auf Constraints, die Constraints bezüglich ihres Informationsgehaltes vergleicht: $\varphi \models_\Gamma \psi$ bedeutet gerade, daß φ mindestens die Information von ψ enthält. Äquivalenz bedeutet gerade, daß zwei Constraints exakt dieselbe Information enthalten. Trotzdem können äquivalente Constraints völlig verschiedene Formen haben. Beispielsweise sind $\mathsf{X} + \mathsf{Y} \doteq \mathsf{5} \wedge \mathsf{X} - \mathsf{Y} \doteq \mathsf{1}$ und $\mathsf{X} \doteq \mathsf{3} \wedge \mathsf{Y} \doteq \mathsf{2}$ in INT äquivalent.

Zu einem gegebenem Constraintsystem $\Gamma = (\Sigma, \Delta, \mathrm{CON})$, können wir die in einem Constraint enthaltenen Informationen als die Menge aller subsumierten Constraints definieren:

$$\mathcal{I}(\varphi) := \{\psi \in \mathrm{CON} \mid \varphi \models_\Gamma \psi\}.$$

Wir haben dann wie gewünscht:

$$\varphi \models_\Gamma \psi \iff \mathcal{I}(\psi) \subseteq \mathcal{I}(\varphi).$$

Ein Constraintsystem $\Gamma = (\Sigma, \Delta, \mathrm{CON})$ heißt **erfüllbarkeitsvollständig** genau dann, wenn

$$\varphi \not\models_\Delta \bot \Rightarrow \Delta \models \exists\, \varphi$$

für alle Constraints $\varphi \in \mathrm{CON}$ gilt.

Proposition 4.1 *Für jedes Constraintsystem $\Gamma = (\Sigma, \Delta, \mathrm{CON})$ sind die folgenden Aussagen äquivalent:*

1. *Γ ist erfüllbarkeitsvollständig.*
2. *für jeden Constraint $\varphi \in \mathrm{CON}$: $\Delta \models \exists\, \varphi$ oder $\Delta \models \neg\exists\, \varphi$.*
3. *für alle Modelle $\mathcal{A}, \mathcal{B}$ von Γ und alle Constraints $\varphi \in \mathrm{CON}$: $\mathcal{A} \models \exists\varphi \iff \mathcal{B} \models \exists\varphi$.*
4. *für alle Modelle $\mathcal{A}, \mathcal{B}$ von Γ und alle Constraints $\varphi \in \mathrm{CON}$: $\mathcal{A} \models \neg\exists\varphi \iff \mathcal{B} \models \neg\exists\varphi$.*
5. *für jedes Modell $\mathcal{A}$ von Γ und jeden Constraint $\varphi \in \mathrm{CON}$: $\mathcal{A} \models \exists\,\varphi \iff \Delta \models \exists\,\varphi$.*
6. *$\Delta \not\models \exists\,\varphi \;\Rightarrow\; \varphi \models_\Delta \bot$*

Erfüllbarkeitsvollständigkeit bedeutet also gerade, daß die Constrainttheorie hinreichend stark ist, die Modelle so einzuschränken, daß sie sich nicht bezüglich der Erfüllbarkeit von Constraints unterscheiden.

4.1 Unabhängigkeit von Constraintsystemen

In diesem Abschnitt wollen wir uns einer weiteren wichtigen Eigenschaft von Constraintsystemen zuwenden.

Definition 4.2 (Unabhängigkeit) *Ein Constraintsystem Γ heißt* **unabhängig** *genau dann, wenn*

$$\varphi \models_\Gamma \bigvee_{i=1}^{n} \psi_i \;\Rightarrow\; \exists i \in \{1, \ldots, n\} :\; \varphi \models_\Gamma \psi_i$$

für alle Constraints φ und $\psi_1, \ldots, \psi_n$ von Γ gilt.

Beispiel 4.2 Das Constraintsystem INT ist **nicht** unabhängig, da beispielsweise

$$\top \models_{\mathrm{INT}} \mathsf{X} \mathrel{\dot<} \mathsf{Y} \vee \mathsf{X} \doteq \mathsf{Y} \vee \mathsf{Y} \mathrel{\dot<} \mathsf{X}$$

gilt, aber keiner der drei atomaren Constraints für sich alleine gilt.

Ein weiteres Beispiel für die Verletzung der Unabhängigkeit ist

$$(\mathsf{X}+2) * (\mathsf{X}+3) \doteq 0 \models_{\mathrm{INT}} \mathsf{X} \doteq -2 \vee \mathsf{X} \doteq -3.$$

Wenn wir aber die Expressivität von INT verringern, indem wir $\dot<$ und $*$ aus der Signatur streichen, erhalten wir ein unabhängiges Constraintsystem. □

Da in der Prädikatenlogik die Äquivalenz

$$\varphi \to (\psi \vee \psi') \mathrel{|\!\!=\!\!|} (\varphi \wedge \neg\psi) \to \psi'$$

gilt, können wir Unabhängigkeit auch mittels Negation und Erfüllbarkeit charakterisieren.

Proposition 4.2 *Eine Constraintssystem Γ ist genau dann unabhängig, wenn für alle Constraints φ und $\psi_1, \ldots, \psi_n$ gilt:*

$$\varphi \wedge \bigwedge_{i=1}^{n} \neg\psi_i \models_\Gamma \bot \;\Rightarrow\; \forall\, i \in \{1, \ldots, n\} :\; \varphi \wedge \neg\psi_i \models_\Gamma \bot.$$

Proposition 4.3 *Sei Γ ein unabhängiges Constraintsystem, dessen Subsumptionsrelation $\models_\Gamma$ auf den Constraints von Γ entscheidbar ist. Dann ist die Subsumptionsrelation auch auf den Booleschen Kombinationen von Constraints von Γ entscheidbar (das heißt auf der Menge aller Formeln, die aus den Constraints von Γ durch Verknüpfung mit den Junktoren $\neg$, $\wedge$ und $\vee$ erhalten werden können).*

Beweis Im folgenden bezeichnen wir eine Konjunktion $\varphi \wedge \bigwedge_{i=1}^{n} \neg\psi_i$ als Klausel, falls φ und $\psi_1, \ldots, \psi_n$ Constraints von Γ sind.

Man sieht leicht ein, daß unter den gemachten Vorraussetzungen die Erfüllbarkeit von Klauseln (d.h. $\sigma \not\models_\Gamma \bot$) entscheidbar ist.

Da jede Boolesche Kombination von Constraints mit den üblichen Regeln für disjunktive Normalform in eine logisch äquivalente Disjunktion von Klauseln überführbar ist, und

$$\sigma \vee \tau \models_\Gamma \bot \iff \sigma \models_\Gamma \bot \text{ und } \tau \models_\Gamma \bot$$

gilt, können wir die Erfüllbarkeit von Booleschen Kombinationen entscheiden. Da zudem

$$\sigma \models_\Gamma \tau \iff \sigma \wedge \neg\tau \models_\Gamma \bot$$

gilt, können wir damit auch die Subsumption von Booleschen Kombinationen entscheiden. $\Box$

Kapitel 5

Der Constraintkalkül K

Wir geben nun einen Kalkül K an, der mit Constraints rechnet. Am Beispiel dieses Kalküls werden wir eine Reihe von wichtigen Operationen auf Constraints studieren können. Für sich alleine ist der Kalkül für praktische Zwecke wenig brauchbar, er wird aber später als wichtiger Bestandteil von praktisch nützlichen Constraintprogrammiersprachen auftauchen. Der Kalkül K ist also eine didaktische Konstruktion, mit der wir eine Reihe von wichtigen und nichttrivialen Konzepten in Reinkultur studieren können.

5.1 Der Kalkül $K(\Gamma)$

Der Kalkül K erweitert ein Constraintsystem Γ um sogenannte **Konditionale**, die wir als logische Version des if–then–else–Konstrukts in Programmiersprachen sehen können. Dabei ist das Constraintsystem Γ beliebig vorgebbar, der Kalkül K ist also in Wirklichkeit eine parameterisierte Konstruktion $K(\Gamma)$, die zu jedem Constraintsystem Γ einen Kalkül $K(\Gamma)$ liefert.

Sei also ein Constraintsystem Γ gegeben. Der entsprechende Kalkül $K(\Gamma)$ ist in Abbildung 5.1 angegeben. Dabei ist zu beachten, daß $[\varphi] \subseteq [\sigma]$ ist und die Konjunktion für Constraints in $[\varphi]$ mit der Konjunktion für Ausdrücke in $[\sigma]$ identisch ist.

Ausdrücke der Form $\langle\varphi;\sigma;\tau\rangle$ heißen **Konditionale** und werden als

$$\textbf{if } \varphi \textbf{ then } \sigma \textbf{ else } \tau \textbf{ fi}$$

gelesen. Die erste Komponente φ ist ein Constraint und wird als **Wächter**[1] des Konditionals bezeichnet.

Eine wichtige Eigenschaft der Kongruenz von $K(\Gamma)$ ist die Tatsache, daß sie in Γ äquivalente Constraints identifiziert.

Beispiel 5.1 (Reduktion in $K(\mathrm{INT})$) Wir betrachten eine Reduktion in $K(\mathrm{INT})$, dabei steht eine Matrix für die Konjunktion ihrer Zeilen:

[1] englisch: **guard**

Syntax

φ, ψ	:	Constraint von Γ
σ, τ	::=	$\varphi \mid \alpha \mid \sigma \wedge \tau$
α, β	::=	$\langle \varphi; \sigma; \tau \rangle \mid \top \mid \alpha \wedge \beta$

Kongruenz

- $([\sigma], \wedge, \top)/\equiv$ ist kommutativer Monoid
- $\varphi \equiv \psi$ falls $\varphi \models\!\mid_\Gamma \psi$

Reduktion

- $\varphi \wedge \langle \psi; \sigma; \tau \rangle \wedge \alpha \longrightarrow \varphi \wedge \sigma \wedge \alpha$ falls $\varphi \models_\Gamma \psi$.
- $\varphi \wedge \langle \psi; \sigma; \tau \rangle \wedge \alpha \longrightarrow \varphi \wedge \tau \wedge \alpha$ falls $\varphi \models_\Gamma \neg\psi$.

Abbildung 5.1: Der Kalkül $K(\Gamma)$

$$
\begin{aligned}
\begin{pmatrix} \mathsf{X} \doteq \mathsf{Y}+1 \wedge \mathsf{X}+\mathsf{Y} \doteq \mathsf{Z} \\ \left\langle 1 \dot{<} \mathsf{Y}; \mathsf{U} \doteq 2*\mathsf{X}; \bot \right\rangle \\ \left\langle \mathsf{Y} \dot{<} \mathsf{X}; \mathsf{Z} \doteq 5; \bot \right\rangle \end{pmatrix}
&\longrightarrow
\begin{pmatrix} \mathsf{X} \doteq \mathsf{Y}+1 \wedge \mathsf{X}+\mathsf{Y} \doteq \mathsf{Z} \\ \left\langle 1 \dot{<} \mathsf{Y}; \mathsf{U} \doteq 2*\mathsf{X}; \bot \right\rangle \\ \mathsf{Z} \doteq 5 \end{pmatrix} \\
&\equiv
\begin{pmatrix} \mathsf{X} \doteq 3 \wedge \mathsf{Y} \doteq 2 \wedge \mathsf{Z} \doteq 5 \\ \left\langle 1 \dot{<} \mathsf{Y}; \mathsf{U} \doteq 2*\mathsf{X}; \bot \right\rangle \end{pmatrix} \\
&\longrightarrow
\begin{pmatrix} \mathsf{X} \doteq 3 \wedge \mathsf{Y} \doteq 2 \wedge \mathsf{Z} \doteq 5 \\ \mathsf{U} \doteq 6 \end{pmatrix}
\end{aligned}
$$

Das Beispiel macht klar, daß Reduktion im allgemeinen nicht nach einer sequentiellen Strategie erfolgen kann: der Wächter $1 \dot{<} \mathsf{Y}$ wird von dem Constraint $\mathsf{X} \doteq \mathsf{Y}+1 \wedge \mathsf{X}+\mathsf{Y} \doteq 2$ nicht **determiniert**, das heißt weder subsumiert noch dissubsumiert. □

5.2 Logische Semantik

Wir können jedem Ausdruck des Kalküls $K(\Gamma)$ eine prädikatenlogische Formel über der Signatur von Σ zuordnen, indem wir das Konditional gemäß

$$\langle \varphi; \sigma; \tau \rangle \;\Rightarrow\; (\varphi \wedge \sigma) \vee (\neg\varphi \wedge \tau)$$

übersetzen. Alternativ können wir das Konditional aber auch als einen neuen Konstruktor für Formeln auffassen, dessen Semantik durch die Äquivalenz

$$\langle\varphi; \sigma; \tau\rangle \models\mid (\varphi \wedge \sigma) \vee (\neg\varphi \wedge \tau)$$

festgelegt ist. Wir werden diese Version benutzen, da wir dann die Ausdrücke des Kalküls wie Formeln verwenden können.

Proposition 5.1 *Sei $\sigma \equiv \tau$ oder $\sigma \longrightarrow \tau$. Dann $\sigma \models\mid_\Gamma \tau$.*

Beweis Für $\sigma \equiv \tau$ ist die Behauptung klar, da $\models\mid_\Gamma$ eine Kongruenz ist, die die Axiome für $\equiv$ erfüllt.

Um die Behauptung für $\sigma \longrightarrow \tau$ zu zeigen, genügt es zu zeigen, daß $\models\mid_\Gamma$ die beiden Axiome für $\longrightarrow$ erfüllt.

Sei $\varphi \models_\Gamma \psi$. Dann gilt $\varphi \wedge \neg\psi \models\mid_\Gamma \bot$ und $\varphi \wedge \psi \models\mid_\Gamma \varphi$, also folglich

$$\begin{array}{rcl} \varphi \wedge ((\psi \wedge \sigma) \vee (\neg\psi \wedge \tau)) & \models\mid & (\varphi \wedge \psi \wedge \sigma) \vee (\varphi \wedge \neg\psi \wedge \tau) \\ & \models\mid_\Gamma & (\varphi \wedge \sigma) \vee (\bot \wedge \tau) \\ & \models\mid & \varphi \wedge \sigma \end{array}$$

Die Erfüllung des zweiten Axioms läßt sich analog zeigen. □

Wir wissen also jetzt, daß Reduktionen in $K(\Gamma)$ eine Äquivalenztransformation für alle Modelle von Γ ist. Diese Aussage kann als eine Art Korrektheitsaussage über den Kalkül angesehen werden.

Dagegen entspricht

$$\sigma \models_\Gamma \bot \;\Rightarrow\; \sigma \longrightarrow^* \bot$$

einer Vollständigkeitsaussage. Vollständigkeit in diesem Sinne ist aber nicht erfüllt, da zum Beispiel in INT das Konditional

$$\left\langle \mathsf{X} \dot{<} \mathsf{1}; \bot; \bot \right\rangle$$

irreduzibel, aber dennoch äquivalent zu $\bot$ ist.

Proposition 5.2 *Die Reduktionsrelation von $K(\Gamma)$ ist terminierend, das heißt, es gibt keine unendliche Kette*

$$\sigma_1 \longrightarrow \sigma_2 \longrightarrow \sigma_3 \longrightarrow \ldots.$$

Man beachte, daß der Kalkül K Negation ausdrücken kann, da

$$\langle\varphi; \bot; \top\rangle \models\mid \neg\varphi.$$

Der Kalkül $K(\Gamma)$ erweitert die Constraints von Γ also insbesondere auch um Negation.

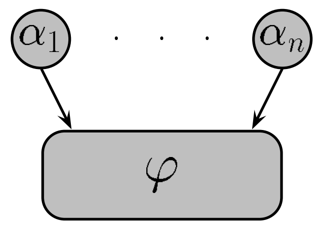


Abbildung 5.2: Mit einer Tafel verbundene Agenten

Proposition 5.3 *Sei Γ ein unabhängiges Constraintsystem. Dann gilt:*

$$\varphi \wedge \bigwedge_{i=1}^{n} \neg\psi_i \models\mid_\Gamma \perp \iff \varphi \wedge \bigwedge_{i=1}^{n} \langle \psi_i; \perp; \top \rangle \longrightarrow^* \perp$$

Beweis Die Richtung „$\Leftarrow$" folgt aus der Tatsache, daß Reduktion in $K(\Gamma)$ eine Äquivalenzumformung in Γ ist. Die andere Richtung folgt unmittelbar aus der Unabhängigkeit. □

Wir können uns den Kalkül K als interaktives Programmiersystem, also als eine abstrakte Maschine, vorstellen. Zu jedem Zeitpunkt repräsentiert die Maschine einen Ausdruck σ modulo Kongruenz. Sie führt Reduktionsschritte $\sigma \longrightarrow \tau$ durch, bis der repräsentierte Ausdruck irreduzibel ist. Die Maschine startet mit dem Ausdruck $\top$. Der Programmierer kann der Maschine beliebige Ausdrücke τ eingeben, die dann zu dem gerade von der Maschine repräsentierten Ausdruck σ hinzu konjungiert werden (d.h. die Maschine repräsentiert nach der Eingabe τ den Ausdruck $\sigma \wedge \tau$).

Damit der Programmierer die Maschine beobachten kann, kann er den repräsentierten Ausdruck in geeigneter Weise inspizieren. Zudem hat die Maschine ein Lämpchen, das leuchtet, solange Reduktionsschritte durchgeführt werden, und das verlöscht, sobald der repräsentierte Ausdruck irreduzibel ist.

Wir können den Kalkül K auch als ein nebenläufiges Programmiersystem sehen. Dazu machen wir uns klar, daß jeder Ausdruck die Form

$$\varphi \wedge \alpha_1 \wedge \ldots \wedge \alpha_n$$

hat, wobei jedes α_i ein Konditional ist. Wir werden φ als Tafel (engl.: blackboard) und die α_i als Agenten bezeichnen, skizziert ist dies in Abbildung 5.2.

Ein Agent $\langle \psi; \sigma; \tau \rangle$ bleibt solange inaktiv, bis die Tafel φ den Wächter ψ determiniert. Das bedeutet also, daß die Agenten auf das Vorhandensein von Informationen auf der Tafel synchronisieren. Sobald genügend Information auf der Tafel vorhanden ist, reduziert $\langle \psi; \sigma; \tau \rangle$ zu σ oder τ. Reduktion bedeutet also, daß der Agent verschwindet und dabei einen Constraint auf die Tafel schreibt, sowie neue Agenten erzeugen kann.

Eine wichtige Eigenschaft von Kalkül K wird als Monotonie bezeichnet. Monotonie bedeutet, daß Information, die einmal auf der Tafel steht, nicht wieder entfernt wird, und daß ein determinierter Wächter determiniert bleibt, wenn zusätzlich Information auf die Tafel geschrieben wird. Dies liegt einfach daran, daß Subsumption die Eigenschaft

$$\varphi \models_\Gamma \psi \;\Rightarrow\; \varphi \wedge \varphi' \models_\Gamma \psi$$

erfüllt.

5.3 Saraswats Tell–Operatoren

In Kalkül K erfolgt das Erweitern der Tafel um neue Constraints implizit: Sobald ein Constraint auf oberster Ebene auftaucht (also nicht als Teil eines Konditionals), ist er Bestandteil der Tafel. Diese Konstruktion abstrahiert erheblich gegenüber einer Implementierung, die neue Constraints natürlich mit einer geeigneten Operation in die Tafel eintragen muß. Um diesen Aspekt genauer beleuchten zu können, erweitern wir den Kalkül K um zwei neue Kombinatoren, die in der Literatur als **eventual tell** und als **atomic tell** bekannt sind.

$$\alpha, \beta \;::=\; \langle\varphi;\sigma;\tau\rangle \quad|\quad \mathsf{etell}\,[\varphi] \quad|\quad \mathsf{atell}\,[\varphi] \quad|\quad \top \quad|\quad \alpha \wedge \beta$$

Die Semantik von $\mathsf{etell}\,[\cdot]$ ist durch dir Reduktionsregel

$$\mathsf{etell}\,[\varphi] \wedge \sigma \longrightarrow \varphi \wedge \sigma$$

gegeben. Reduktion von $\mathsf{etell}\,[\varphi]$ bedeutet also gerade, daß der Constraint φ auf die Tafel geschrieben wird.

Man macht sich leicht klar, daß sich Kalkül K nicht wesentlich ändert, wenn man ihn nur um $\mathsf{etell}\,[\cdot]$ erweitert. Wir werden aber gleich sehen, daß in Gegenwart des $\mathsf{atell}\,[\cdot]$–Kombinators ein signifikanter Unterschied zwischen φ und $\mathsf{etell}\,[\varphi]$ besteht.

Die Semantik von atomic tell ist durch zwei Regeln gegeben:

$$\begin{array}{lcll} \varphi \wedge \mathsf{atell}\,[\psi] \wedge \alpha & \longrightarrow & \varphi \wedge \psi \wedge \alpha & \text{falls } \varphi \wedge \psi \text{ erfüllbar in } \Gamma \\ \varphi \wedge \mathsf{atell}\,[\psi] \wedge \alpha & \longrightarrow & \varphi \wedge \alpha & \text{falls } \varphi \wedge \psi \text{ unerfüllbar in } \Gamma \end{array}$$

Im Gegensatz zu $\mathsf{etell}\,[\cdot]$ schreibt $\mathsf{atell}\,[\cdot]$ seinen Constraint nur dann auf die Tafel, wenn die erweiterte Tafel erfüllbar ist. Ist dies nicht möglich, gibt $\mathsf{atell}\,[\cdot]$ auf und läßt die Tafel unverändert.

Beispiel 5.2 Man betrachte den Ausdruck

$$\mathsf{atell}\,[\mathsf{X} \doteq 1] \wedge \mathsf{atell}\,[\mathsf{X} \doteq 2]$$

Dann sind zwei maximale Reduktionen möglich:

1. $\mathsf{atell}\,[\mathsf{X} \doteq 1] \wedge \mathsf{atell}\,[\mathsf{X} \doteq 2] \longrightarrow \mathsf{X} \doteq 1$
2. $\mathsf{atell}\,[\mathsf{X} \doteq 1] \wedge \mathsf{atell}\,[\mathsf{X} \doteq 2] \longrightarrow \mathsf{X} \doteq 2$

□

Damit ist klar, daß der um $\mathsf{atell}\,[\cdot]$ erweiterte Kalkül völlig neue Eigenschaften hat.

Zum einen ist eine logische Semantik nicht mehr möglich, da dies im obigen Beispiel die Äquivalenz von $\mathsf{X} \doteq 1$ und $\mathsf{X} \doteq 2$ zur Folge hätte. Der erweiterte Kalkül ist also aus der bisher eingenommenen logischen Sicht inkonsistent. Aus operationaler Sicht würde man dagegen von **Indeterminismus** sprechen: Je nachdem in welcher Reihenfolge die Maschine die $\mathsf{atell}\,[\cdot]$–Agenten ausführt, sind verschiedene Ergebnisse möglich.

Im Gegensatz zu Konditionalen ist der $\mathsf{atell}\,[\cdot]$–Operator nicht monoton: Wenn

$$\varphi \wedge \mathsf{atell}\,[\psi] \longrightarrow \varphi \wedge \psi$$

gilt, braucht

$$\varphi \wedge \varphi' \wedge \mathsf{atell}\,[\psi] \longrightarrow \varphi \wedge \varphi' \wedge \psi$$

noch lange nicht zu gelten. Das bedeutet also, daß mit Fortschreiten der Berechnung vorher mögliche $\mathsf{atell}\,[\cdot]$–Reduktionen ungültig werden können.

Mit Hilfe von $\mathsf{atell}\,[\cdot]$ ist es möglich, die Tafel immer erfüllbar zu halten, also einen System–Crash

$$\perp \wedge \sigma \longrightarrow^* \perp$$

zu verhindern.

Proposition 5.4 *Sei σ ein Ausdruck, der kein $\mathsf{etell}\,[\cdot]$ enthält, und in dem Constraints nur als Wächter von Konditionalen und Argumente von $\mathsf{atell}\,[\cdot]$ vorkommen. Dann reduziert σ nur auf Ausdrücke mit erfüllbarer Tafel, das heißt $\sigma \not\longrightarrow^* \perp$.*

Wie bereits angekündigt, gibt es in Kombination mit $\mathsf{atell}\,[\cdot]$ einen signifikanten Unterschied zwischen φ und $\mathsf{etell}\,[\varphi]$. Beispielweise kann

$$\mathsf{etell}\,[\mathsf{X} \doteq 1] \wedge \mathsf{atell}\,[\mathsf{X} \doteq 2]$$

entweder zu $\mathsf{X} \doteq 1$ oder zu $\perp$ reduzieren. Dagegen kann

$$\mathsf{X} \doteq 1 \wedge \mathsf{atell}\,[\mathsf{X} \doteq 2]$$

nur zu $\mathsf{X} \doteq 1$ reduzieren.

Kapitel 6

Die Systeme CET und HER

6.1 Das Constraintsystem CET

Wir betrachten nun ein klassisches Constraintsystem, das eng mit der Unifikation von Termen zusammenhängt.

Im folgenden sei Σ eine prädikatenlogische Signatur, über der unendlich viele geschlossene Terme gebildet werden können und die keine Relationssymbole enthält. Das bedeutet, daß Σ mindestens eine Konstante enthalten muß. Wenn Σ endlich ist, muß Σ zudem mindestens ein einstelliges oder mehrstelliges Funktionssymbol enthalten. Im folgenden werden x, y, z stets Variablen und s, t, u, v stets Terme bezeichnen.

Sei V eine Menge von Variablen. Die Termalgebra $\mathcal{T}(\Sigma, V)$ ist diejenige Struktur für Σ, deren Universum aus allen Termen besteht, die man über Σ und V bilden kann, und die die Funktionssymbole $f \in \Sigma$ wie folgt interpretiert:

$$\mathcal{T}(\Sigma, V)\,[\![f]\!]\,(s_1, \ldots, s_n) := f(s_1, \ldots, s_n).$$

Man sieht, daß die Variablenbelegungen für $\mathcal{T}(\Sigma, V)$ gerade die Substitutionen sind, die Variablen auf (Σ, V)–Terme abbilden. Im folgenden bezeichnen wir mit $\mathcal{H}(\Sigma) := \mathcal{T}(\Sigma, \emptyset)$ die sogenannte **Herbrand**- oder **Grundtermalgebra** für Σ.

Mit $\mathcal{T}(\Sigma) := \mathcal{T}(\Sigma, \mathit{VAR})$ bezeichnen wir die **volle Termalgebra**, die über der Menge VAR aller Variablen gebildet ist.

Proposition 6.1 *Sei V eine Menge von Variablen, θ eine Belegung für $\mathcal{T}(\Sigma, V)$, und seien s, t Terme über Σ. Dann gilt:*

1. $\mathcal{T}(\Sigma, V), \theta \models s \doteq t \iff \theta s = \theta t.$

2. *s und t sind unifizierbar genau dann, wenn die Gleichung $s \doteq t$ in $\mathcal{T}(\Sigma, V)$ erfüllbar ist.*

3. *Eine Substitution ρ für Σ ist ein Unifikator von s und t genau dann, wenn ρ eine Lösung der Gleichung $s \doteq t$ in $\mathcal{T}(\Sigma)$ ist.*

Das Constraintsystem CET(Σ) ist nun wie folgt gegeben: Die Signatur von CET(Σ) ist Σ, die Theorie von CET(Σ) ist die Menge aller Sätze, die als Instanzen der Axiomenschemata

$$
\begin{array}{lll}
\text{(Ax1)} & \forall\,(f(x_1,\dots,x_n) \doteq f(y_1,\dots,y_n) \to \bigwedge_{i=1}^n x_i \doteq y_i) & \\
\text{(Ax2)} & \forall\,(f(x_1,\dots,x_n) \doteq g(y_1,\dots,y_n) \to \bot) & (f \neq g) \\
\text{(Ax3)} & \forall\,(x \doteq f(s_1,\dots,s_n) \to \bot) & (x \in \mathcal{V}(s_1) \cup \dots \cup \mathcal{V}(s_n))
\end{array}
$$

erhalten werden können, und die Constraints von CET(Σ) sind durch

$$\varphi, \psi \;::=\; s \doteq t \quad|\quad \bot \quad|\quad \top \quad|\quad \varphi \wedge \psi$$

gegeben.

Proposition 6.2 *Für jede Menge V von Variablen ist $\mathcal{T}(\Sigma, V)$ ein Modell von* CET(Σ).

Übung 6.1 *Zeige, daß $\mathcal{T}(\Sigma, V)$ das dritte Axiomenschema der Theorie von* CET(Σ) *erfüllt.*

Da die Signatur im folgenden stets festgelegt ist, werden wir abkürzend $\mathcal{T}(V)$, $\mathcal{H}$, $\mathcal{T}$ und CET schreiben.

Die Bezeichnung CET ist ein Akronym für „Clark's Equality Theory“. Clark hat die drei Axiomenschemata von CET im Zusammenhang mit der Semantik von Negation als Failure angegeben [Cla78].

In der Literatur wird das Constraintsystem CET oft auch als „Herbrand“ oder „Finite Tree System“ bezeichnet.

6.2 Ein Simplifikationssystem für CET

Unser Ziel ist ein Algorithmus, der Subsumption und Dissubsumption in CET entscheidet. Grundlage für diesen Algorithmus ist ein System von sogenannten **Simplifikationsregeln**, mit denen Constraints in eine äquivalente Form überführt werden können. Das dabei zugrundeliegende Prinzip ist dasselbe wie beim Lösen von linearen Gleichungssystemen mit dem Gaußschen Eliminationsverfahren.

Abbildung 6.1 definiert einen Kalkül, den wir als Simplifikationssystem HER bezeichen werden. Die Syntax von HER ist gerade die von CET.

Mit $\varphi[s/x]$ bezeichnen wir den Constraint, den man aus φ erhält, indem man alle Auftreten von x durch den Term s ersetzt. Mit unserer bereits definierten Notation für Substitutionen und deren Anwendung haben wir

$$[x/s]\varphi = \varphi[s/x].$$

Da logische Äquivalenz $\models\!\mid$ die Kongruenzaxiome von HER erfüllt, haben wir

$$\varphi \equiv \psi \;\Rightarrow\; \varphi \models\!\mid \psi \;\Rightarrow\; \varphi \models\!\mid_{\text{CET}} \psi.$$

Syntax

x, y, z	:	*Variable*
s, t, u	:	*Terme über* Σ
φ, ψ	::=	$s \doteq t \quad \mid \quad \bot \quad \mid \quad \top \quad \mid \quad \varphi \wedge \psi$

Kongruenz

- $([\varphi], \wedge, \top)/\equiv$ ist kommutativer Monoid
- $s \doteq t \;\equiv\; t \doteq s$

Reduktion

- $x \doteq x \wedge \varphi \longrightarrow \varphi$ (6.1)
- $\bot \wedge \varphi \longrightarrow \bot$ falls $\varphi \not\equiv \top$ (6.2)
- $x \doteq y \wedge \varphi \longrightarrow x \doteq y \wedge \varphi[y/x]$ falls $x \neq y$ und $x, y \in \mathcal{V}(\varphi)$ (6.3)
- $x \doteq f(\overline{s}) \wedge \varphi \longrightarrow x \doteq f(\overline{s}) \wedge \varphi[f(\overline{s})/x]$ falls $x \in \mathcal{V}(\varphi)$ und $x \notin \mathcal{V}(\overline{s})$ (6.4)
- $f(\overline{s}) \doteq f(\overline{t}) \wedge \varphi \longrightarrow \overline{s} \doteq \overline{t} \wedge \varphi$ (6.5)
- $f(\overline{s}) \doteq g(\overline{t}) \wedge \varphi \longrightarrow \bot$ falls $f \neq g$ (6.6)
- $x \doteq f(\overline{s}) \wedge \varphi \longrightarrow \bot$ falls $x \in \mathcal{V}(\overline{s})$ (6.7)

Abbildung 6.1: Das Simplifikationssystem HER

Die ersten vier Axiome für die Reduktionsrelation werden ebenfalls von der logischen Äquivalenz $\models\mid$ erfüllt. Die restlichen drei Axiome für die Reduktionsrelation entsprechen gerade den Axiomenschemata von CET. Damit haben wir

$$\varphi \longrightarrow \psi \;\Rightarrow\; \varphi \models\mid_{\mathrm{CET}} \psi.$$

Definition 6.1 (Eliminierte Variable) *Eine Variable x heißt* **eliminiert in** *φ genau dann, wenn es einen Term s und einen Constraint φ' gibt, so daß $\varphi \equiv x \doteq s \wedge \varphi'$ und x weder in s noch in φ' vorkommt.*

Proposition 6.3 *Die Reduktionsrelation $\longrightarrow$ von* HER *ist terminierend, das heißt es gibt keine unendliche Ableitung $\varphi_1 \longrightarrow \varphi_2 \longrightarrow \varphi_3 \longrightarrow \cdots$.*

Beweis Wir führen den Beweis durch Widerspruch.

Sei also eine unendliche Ableitung

$$\varphi_1 \longrightarrow \varphi_2 \longrightarrow \varphi_3 \longrightarrow \cdots$$

gegeben.

Da $\perp$ offensichtlich nicht reduzibel ist, können die Regeln (6.2), (6.6) und (6.7) dabei nicht zur Anwendung kommen.

Nun überzeuge man sich davon, daß die Regeln (6.3) und (6.4) die Anzahl der eliminierten Variablen erhöhen, und daß die verbleibenden Regeln diese Anzahl **nicht** herabsetzen. Da keine Regel neue Variablen hinzufügt, bedeutet dies, daß es eine unendliche Ableitung gibt, die nur die Regeln (6.1) und (6.5) benutzt.

Dies ist aber ein Widerspruch, denn man sieht sofort, daß sich jede dieser beiden Regeln nur endlich oft anwenden läßt (Beide verringern die Anzahl Von Auftreten von Variablen oder Funktionssymbolen). □

Ein Constraint von CET heißt **gelöst**, wenn er die Form

$$x_1 \doteq s_1 \wedge \cdots \wedge x_n \doteq s_n \tag{6.8}$$

hat, wobei $n \geq 0$ und die Variablen $x_1, \ldots, x_n$ paarweise verschieden sind und nicht in den Termen $s_1, \ldots, s_n$ vorkommen.

Einem gelösten Constraint (6.8) kann man die Substitution

$$[x_1/s_1, \ldots, x_n/s_n]$$

zuordnen.

Sei φ ein gelöster Constraint in CET und θ die ihm so zugeordnete Substitution. Man überzeugt sich leicht, daß $\theta\varphi$ eine Konjunktion von trivialen Gleichungen $s \doteq s$ ist.

Also haben wir:

Proposition 6.4 *Sei φ ein gelöster Constraint von* CET. *Dann gilt für jede Struktur $\mathcal{A}$ für Σ:*

$$\mathcal{A} \models \exists\varphi.$$

Proposition 6.5 *Sei $\varphi \neq \perp$ ein Constraint von* CET. *Dann ist φ irreduzibel bezüglich* HER *genau dann, wenn φ gelöst ist.*

Beweis Man überzeugt sich leicht davon, daß sich keine der Regeln (6.1)–(6.7) von HER auf eine gelöste Form mehr anwenden läßt. Umgekehrt überzeugt man sich, daß wenn keine der Regeln anwendbar ist, es sich tatsächlich um eine gelöste Form handelt. □

Wir wissen nun, daß wir mit Hilfe des Simplifikationssystems HER die Erfüllbarkeit von Constraints in CET entscheiden können. Dabei gehen wir wie folgt vor. Sei ein Constraint φ gegeben. Von φ ausgehend wenden wir solange Reduktionsregeln an, bis wir einen irreduziblen Constraint ψ erhalten (hier benötigen wir die Terminierung von HER nach Proposition 6.3).

Da $\longrightarrow \subseteq \models\mid_{\mathrm{CET}}$, haben wir

$$\varphi \models\mid_{\mathrm{CET}} \psi,$$

das heißt, ψ ist genau dann erfüllbar, wenn φ erfüllbar ist.

Aufgrund der beiden vorhergehenden Propositionen ist aber ψ genau dann erfüllbar, wenn es verschieden von $\perp$ ist.

Seien φ und ψ Constraints von CET. Dann heißt ψ **eine gelöste Form für** φ, falls ψ ein gelöster Constraint ist, und

$$\varphi \models\!\!\mid_{\mathrm{CET}} \psi$$

gilt.

Proposition 6.6 *Sei $\varphi \longrightarrow^* \psi$ und sei ψ irreduzibel bezüglich* HER. *Dann ist φ genau dann unerfüllbar, wenn $\psi = \perp$. Wenn $\psi \neq \perp$, dann ist ψ eine gelöste Form für φ. Also hat jeder erfüllbare Constraint eine gelöste Form.*

Proposition 6.7 CET *ist erfüllbarkeitsvollständig.*

Beweis Erfüllbarkeitsvollständigkeit bedeutet, daß ein Constraint, der in einem Modell von CET erfüllbar ist, in jedem Modell von CET erfüllbar ist (siehe Proposition 4.1).

Dies folgt aber sofort aus der Tatsache, daß jeder erfüllbare Constraint zu einem gelösten Constraint äquivalent ist, und daß jeder gelöste Constraint in jeder Struktur erfüllbar ist (siehe Proposition 6.4). □

6.3 Unifikation

In diesem Abschnitt werden wir uns noch einmal der Unifikation von Termen und dem Zusammenhang zum System HER widmen. Wir setzen hier die Begriffe voraus, die bereits in Abschnitt 2.4 ab Seite 14 eingeführt wurden.

Proposition 6.8 *Die Terme s und t sind genau dann unifizierbar, wenn die Gleichung $s \doteq t$ in* CET *erfüllbar ist.*

Beweis Mit Proposition 6.1 (3.) folgt, daß s und t genau dann unifizierbar sind, wenn $s \doteq t$ in $\mathcal{T}$ erfüllbar ist.

Da $\mathcal{T}$ ein Modell von CET ist, und CET erfüllbarkeitsvollständig ist, ist $s \doteq t$ in $\mathcal{T}$ genau dann erfüllbar, wenn es in CET erfüllbar ist. □

Proposition 6.9 *Sei $s \doteq t \longrightarrow^* \varphi$ und sei $\varphi = x_1 \doteq s_1 \wedge \cdots \wedge x_n \doteq s_n$ gelöst. Dann ist die φ zugeordnete Substitution*

$$\theta = [x_1/s_1, \ldots, x_n/s_n]$$

ein prinzipaler Unifikator von $s \doteq t$.

Beweis Da φ aus $s \doteq t$ mit den Regeln von HER gewonnen werden kann, gilt

$$\mathcal{D}om(\theta) \cup \mathcal{V}(\mathcal{R}an(\theta)) = \mathcal{V}(\varphi) \subseteq \mathcal{V}(s \doteq t).$$

Die Idempotenz von θ rechnet man leicht nach.

Da $\mathcal{T}, \theta \models \varphi$ und $\varphi \models\!|_{\mathrm{CET}}\ s \doteq t$, gilt $\mathcal{T}, \theta \models s \doteq t$ und folglich ist θ ein Unifikator von $s \doteq t$.

Sei nun ρ ein Unifikator von $s \doteq t$. Dann ist $\mathcal{T}, \rho \models s \doteq t$ und folglich $\mathcal{T}, \rho \models \varphi$. Also

$$\rho x_i = \rho s_i = \rho\theta x_i. \qquad (1 \leq i \leq n)$$

Also $\rho = \rho\theta$, was zeigt, daß θ ein allgemeinster Unifikator von $s \doteq t$ ist. □

Abschließen wollen wir diesen Abschnitt mit einem Beispiel, das die Komplexität unseres Verfahrens beleuchtet.

Beispiel 6.2 Betrachte den Constraint

$$\mathsf{X}_1 \doteq \mathsf{f}(\mathsf{X}_2, \mathsf{X}_2) \wedge \mathsf{X}_2 \doteq \mathsf{f}(\mathsf{X}_3, \mathsf{X}_3) \wedge \cdots \wedge \mathsf{X}_{n-1} \doteq \mathsf{f}(\mathsf{X}_n, \mathsf{X}_n).$$

Dann liefert Reduktion mit HER eine gelöste Form, die exponentiell größer ist als der Ausgangsconstraint, da X_1 an einen vollen binären Baum der Tiefe $n-1$ gebunden wird. Das bedeutet, daß ein naiver, auf HER basierender Algorithmus mindestens exponentielle Komplexität hat. □

Paterson und Wegmann [PW78] haben aber gezeigt, daß sich Erfüllbarkeit in CET in linearer Zeit entscheiden läßt. Wir werden später einen quasi–linearen Algorithmus für dieses Problem angeben.

6.4 Subsumption für CET

Ziel in disem Abschnitt wird es sein, ein Entscheidungsverfahren für die Subsumptionsrelation von CET anzugeben. Denn ersten Schritt auf dem Weg zum Ziel liefert dazu das folgende Lemma.

Lemma 6.10 *Sei φ ein gelöster Constraint und θ die zugehörige Substitution. Dann gilt*

$$\varphi \models_{\mathrm{CET}} \psi \iff \mathrm{CET} \models \theta\psi$$

für jede quantorenfreie Formel ψ.

Beweis Bevor wir die beiden Richtungen der Aussage dieses Lemmas beweisen, wollen wir die zu beweisenden Aussagen noch leicht umformen.

Für die linke Seite erhalten wir:

$$\begin{aligned} \varphi \models_{\mathrm{CET}} \psi & \iff \varphi \models\!\mid_{\mathrm{CET}} \varphi \wedge \psi \models\!\mid \varphi \wedge \theta\psi \\ & \iff \varphi \models_{\mathrm{CET}} \theta\psi \end{aligned} \tag{6.9}$$

Für die rechte:

$$\mathrm{CET} \models \theta\psi \iff \theta\psi \models\!\mid_{\mathrm{CET}} \top \tag{6.10}$$

$\boxed{\Leftarrow}$ Diese Richtung ist trivial. Gelte $\theta\psi \models\!\mid_{\mathrm{CET}} \top$, also (6.10). Da $\varphi \models_{\mathrm{CET}} \top$ gilt, gilt natürlich erst recht auch $\varphi \models_{\mathrm{CET}} \theta\psi$, das heißt (6.9).

$\boxed{\Rightarrow}$ Sei nun $\varphi \models_{\mathrm{CET}} \theta\psi$.
Sei $\mathcal{A}$ ein Modell von CET, α eine Variablenbelegung für $\mathcal{A}$ und $\varphi = x_1 \doteq s_1 \wedge \cdots \wedge x_n \doteq s_n$. Wir konstruieren eine Belegung β für $\mathcal{A}$ wie folgt:

$$\beta(x) := \begin{cases} \mathcal{A}[\![\alpha \rhd s_i]\!] & \text{falls } \exists i \in \{1,\ldots,n\}: \; x = x_i \\ \alpha(x) & \text{sonst} \end{cases}$$

Da $\theta\psi$ keine der Variablen $x_1,\ldots,x_n$ enthält, genügt es zu zeigen, daß

$$\mathcal{A}, \beta \models \varphi.$$

Dies ist jedoch der Fall, da, für jedes $i \in \{1,\ldots,n\}$, s_i keine der Variablen $x_1,\ldots,x_n$ enthält und daher

$$\begin{aligned} \mathcal{A}[\![\beta \rhd x_i]\!] &= \beta(x_i) \\ &= \mathcal{A}[\![\alpha \rhd s_i]\!] \quad \text{(Definition von } \beta) \\ &= \mathcal{A}[\![\beta \rhd s_i]\!] \quad (\varphi \text{ gelöst}) \end{aligned}$$

gilt.

□

Wir zeigen nun, wie sich Subsumption von Constraints in CET entscheiden läßt.

Seien φ und ψ gegeben. Um zu entscheiden, ob $\varphi \models_{\mathrm{CET}} \psi$, reduzieren wir φ zunächst mit HER. Wenn sich φ dabei als unerfüllbar herausstellt, gilt

$$\varphi \models_{\mathrm{CET}} \psi$$

trivialerweise.

Ansonsten erhalten wir eine gelöste Form $\overline{\varphi}$ von φ. Sei θ die zugeordnete Substitution. Mit Lemma 6.10 folgt

$$\begin{aligned} \varphi \models_{\mathrm{CET}} \psi & \iff \overline{\varphi} \models_{\mathrm{CET}} \psi \\ & \iff \mathrm{CET} \models \theta\psi. \end{aligned}$$

Nun reduzieren wir $\theta\psi$ mit HER. Wenn sich dabei $\theta\psi$ als unerfüllbar herausstellt, haben wir

$$\varphi \models_{\mathrm{CET}} \neg\psi,$$

da $\varphi \wedge \psi \models\!\mid_{\mathrm{CET}} \overline{\varphi} \wedge \theta\psi$.
Ansonsten erhalten wir eine gelöste Form $\overline{\psi}$ von $\theta\psi$ und haben

$$\varphi \models_{\mathrm{CET}} \psi \iff \top \models_{\mathrm{CET}} \overline{\psi}.$$

Das verbleibende Problem ist sehr einfach und wird von der folgenden Proposition gelöst.

Proposition 6.11 *Sei φ ein gelöster Constraint von* CET*. Dann gilt:*

$$\mathrm{CET} \models \varphi \iff \varphi = \top.$$

Beweis Wir führen den Beweis durch Widerspruch.

Sei dazu $\varphi \neq \top$ ein gelöster Constraint mit $\top \models_{\mathrm{CET}} \varphi$. O.B.d.A. können wir annehmen, daß $\varphi = x \doteq s$ mit $x \notin \mathcal{V}(s)$.

Sei θ eine Substitution mit $\theta x \neq \theta s$. Also

$$\mathcal{T}, \theta \not\models \varphi.$$

Da $\mathcal{T}$ ein Modell von CET ist, haben wir $\top \not\models_{\mathrm{CET}} \varphi$, was unserer Annahme widerspricht. □

Betrachten wir noch einmal die ganze Argumentation, so können wir simultan Subsumption und Dissubsumption entscheiden. Zusammengefaßt ist dies in Abbildung 6.2 für den Fall „$\varphi \models_{\mathrm{CET}} \psi$?“ dargestellt.

6.5 Unabhängigkeit von CET

In diesem Abschnitt wollen wir die Unabhängigkeit des Constraintsystems CET beweisen. Um das Ergebnis zu formulieren brauchen wir etwas technische Vorbereitung.

Die Auswertung $\mathcal{A}[\![\alpha \rhd s]\!]$ eines geschlossenen Termes s in einer Struktur $\mathcal{A}$ unter der Variablenbelegung α hängt nicht von der Belegung α ab. Dem werden wir dadurch Rechnung tragen, daß wir $\mathcal{A}[\![\alpha \rhd s]\!]$ im folgenden einfach als $\mathcal{A}[\![s]\!]$ schreiben.

Proposition 6.12 *Sei $\mathcal{A}$ ein Modell von* CET *und seien s und t geschlossene Terme. Dann gilt:*

$$\mathcal{A}[\![s]\!] = \mathcal{A}[\![t]\!] \iff s = t.$$

Beweis Die Richtung „ $\Leftarrow$ “ ist trivial. Die andere Richtung folgt durch Induktion über s unter Ausnutzung der ersten zwei Axiomenschemata von CET. □

Lemma 6.13 *Sei $\mathcal{A}$ ein Modell von* CET *und seien α und β Belegungen für $\mathcal{A}$. Dann gilt für jeden Term s:*

$$\mathcal{A}[\![\alpha \rhd s]\!] = \mathcal{A}[\![\beta \rhd s]\!] \Rightarrow \forall x \in \mathcal{V}(s): \alpha(x) = \beta(x).$$

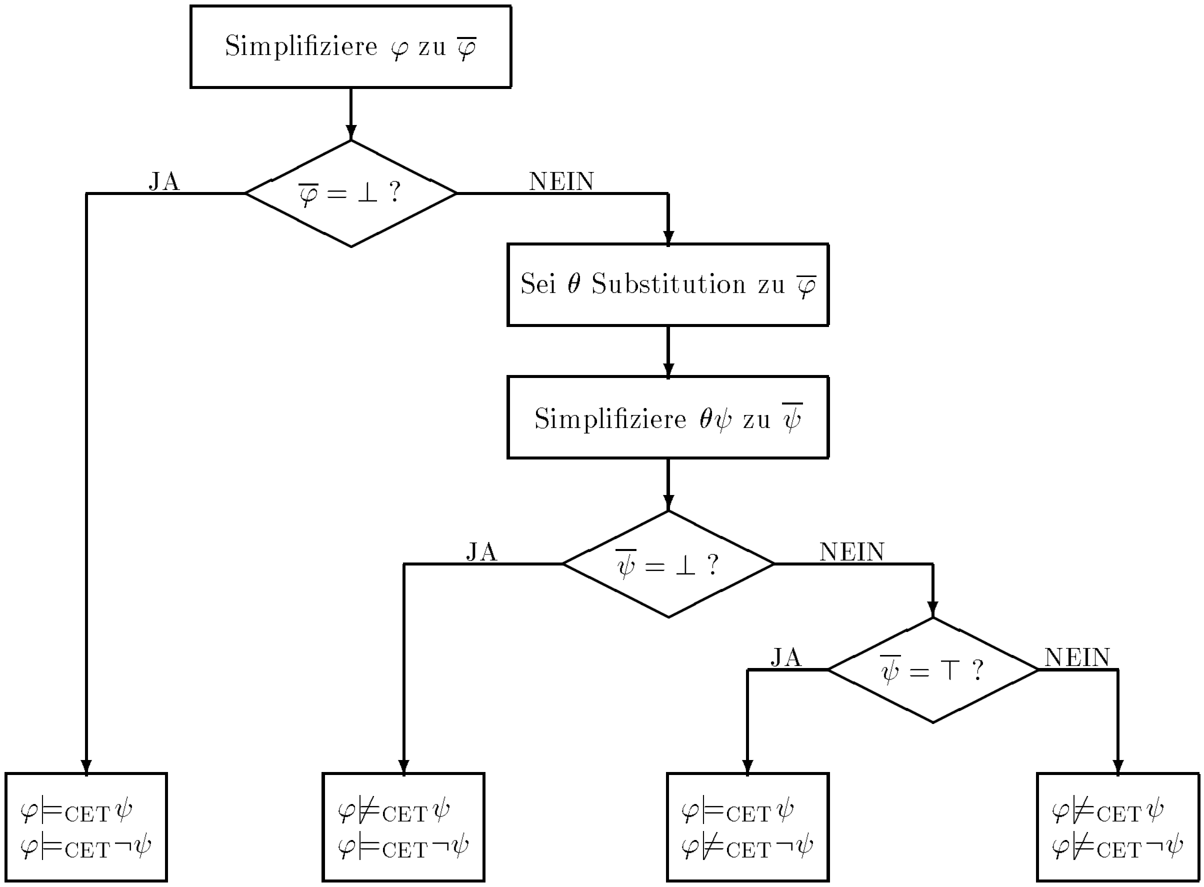


Abbildung 6.2: Entscheiden von Subsumption und Dissubsumption in CET

Beweis Durch Induktion über s.

Wenn s eine Konstante oder eine Variable ist, dann ist die Behauptung trivial.

Ansonsten sei $s = f(s_1, \ldots, s_n)$. Aufgrund des ersten Axiomenschemas von CET folgt $\mathcal{A}[\![\alpha \rhd s_i]\!] = \mathcal{A}[\![\beta \rhd s_i]\!]$ für $i = 1, \ldots, n$. Da jede Variable in s in einem der Unterterme s_i vorkommt, folgt die Behauptung aus der Induktionsvorraussetzung. □

Lemma 6.14 *Sei φ ein gelöster Constraint, $x \in \mathcal{V}(\varphi)$, $\mathcal{A}$ ein Modell von* CET *und α eine Belegung für $\mathcal{A}$. Dann gibt es höchstens einen geschlossenen Term s, so daß*

$$\mathcal{A}, \alpha \models \varphi[s/x].$$

Beweis Seien s und t geschlossene Terme, so daß $\mathcal{A}, \alpha \models \varphi[s/x]$ und $\mathcal{A}, \alpha \models \varphi[t/x]$. Wir zeigen, daß $s = t$ gilt.

Seien β und γ Belegungen mit $\beta(x) = \mathcal{A}[\![s]\!]$ und $\gamma(x) = \mathcal{A}[\![t]\!]$, die sich von α nur an der Stelle x unterscheiden. Wegen Proposition 6.12 genügt es zu zeigen, daß $\beta(x) = \gamma(x)$.

Da x in φ vorkommt, können wir in φ eine Gleichung $y \doteq u$ wählen, so daß entweder $x = y$ oder $x \in \mathcal{V}(u)$. Da φ gelöst ist, kommt x nur auf einer Seite der Gleichung $y \doteq u$ vor.

1. $\boxed{x = y}$: Dann gilt

$$\beta(x) = \mathcal{A}[\![\beta \rhd u]\!] = \mathcal{A}[\![\gamma \rhd u]\!] = \gamma(x),$$

da x in u nicht vorkommt.

2. $\boxed{x \in \mathcal{V}(u)}$: Dann ist $x \neq y$ und folglich

$$\mathcal{A}[\![\beta \rhd u]\!] = \beta(y) = \gamma(y) = \mathcal{A}[\![\gamma \rhd u]\!].$$

Also folgt mit Lemma 6.13, daß $\beta(x) = \gamma(x)$.

□

Das nächste Lemma stammt vom russischen Logiker A. I. Mal'cev [Mal71].

Lemma 6.15 (Mal'cevs Lemma) *Seien $\varphi_1, \ldots, \varphi_n$ gelöste Constraints, so daß jedes φ_i mindestens eine Variable aus $\overline{x}$ enthält. Dann gilt*

$$\text{CET} \models \exists \overline{x} \bigwedge_{i=1}^{n} \neg \varphi_i. \tag{6.11}$$

Beweis Seien $\psi_1, \ldots, \psi_n$ gelöste Constraints, die alle die Variable y enthalten. Dann folgt aus Lemma 6.14 und der Annahme, daß über Σ unendlich viele geschlossene Terme gebildet werden können (siehe dazu den Anfang dieses Kapitels), daß

$$\text{CET} \models \exists x \bigwedge_{i=1}^{n} \neg \psi_i \tag{6.12}$$

gilt.

Den Beweis der Aussage (6.11) führen wir mit Induktion über die Länge von $\overline{x}$. Wenn $\overline{x}$ die Länge null hat, ist die Aussage trivial.

Sei also $\overline{x} = y\overline{y}$. Dann gilt

$$\begin{aligned}
\exists \overline{x} \bigwedge_{i=1}^{n} \neg \varphi_i &= \exists y \exists \overline{y} \bigwedge_{i=1}^{n} \neg \varphi_i \\
&\models\!\mid \exists \overline{y} \exists y \bigwedge_{i=1}^{n} \neg \varphi_i \\
&= \exists \overline{y} \exists y \left(\bigwedge_{j=1}^{n_1} \neg \psi_j \wedge \bigwedge_{j=1}^{n_2} \neg \psi'_j \right)
\end{aligned}$$

Dabei sind die ψ_j die φ_i, die y enthalten (nach Vorraussetzung gibt es mindestens ein derartiges φ_i) und die ψ'_j die φ_i, die y nicht enthalten, also $n = n_1 + n_2$.

Weiterhin gilt

$$\begin{aligned}
\exists \overline{y} \exists y \left(\bigwedge_{j=1}^{n_1} \neg\psi_j \wedge \bigwedge_{j=1}^{n_2} \neg\psi'_j \right) \quad &\models\!\mid \quad \exists \overline{y} \left(\exists y \bigwedge_{j=1}^{n_1} \neg\psi_j \wedge \bigwedge_{j=1}^{n_2} \neg\psi'_j \right) \\
&\models\!\mid_{\mathrm{CET}} \quad \exists \overline{y} \left(\top \wedge \bigwedge_{j=1}^{n_2} \neg\psi'_j \right) \qquad \text{(wegen (6.12))} \\
&\models\!\mid \quad \exists \overline{y} \bigwedge_{j=1}^{n_2} \neg\psi'_j
\end{aligned}$$

Wenn $n_2 > 0$ ist, so ist die letzte Formel nach Induktionsvorraussetzung in CET äquivalent zu $\top$, sonst sowieso gleich $\top$.

Insgesamt haben wir

$$\exists \overline{x} \bigwedge_{i=1}^{n} \neg\varphi_i \models\!\mid_{\mathrm{CET}} \top,$$

also gerade die Aussage (6.11). □

Jetzt sind wir in der Lage, das gewünschte Ergebnis zu formulieren und zu beweisen.

Satz 6.16 (Unabhängigkeit von CET**)** *Das Constraintsystem* CET *ist unabhängig.*

Beweis Sei

$$\varphi \models_{\mathrm{CET}} \bigvee_{i=1}^{n} \psi_i.$$

Wir müssen zeigen, daß $\varphi \models \psi_i$ für mindestens ein $i \in \{1, \ldots, n\}$ gilt.

Da jeder Constraint von CET entweder zu $\perp$ oder einer gelösten Form simplifiziert werden kann, können wir o.B.d.A. annehmen, daß φ und $\psi_1, \ldots, \psi_n$ gelöst sind.

Sei θ die φ zugeordnete Substitution. Mit Lemma 6.10 haben wir

$$\mathrm{CET} \models \bigvee_{i=1}^{n} \theta\psi_i.$$

Mit Malc'evs Lemma 6.15 folgt jetzt, daß es ein $i \in \{1, \ldots, n\}$ mit $\mathrm{CET} \models \theta\psi_i$ gibt.

Also folgt wiederum mit Lemma 6.10, daß

$$\varphi \models_{\mathrm{CET}} \psi_i.$$

□

Mit dem letzten Satz und dem im vorherigen Abschnitt entwickelten Entscheidungsverfahren für die Subsumptionsrelation zwischen Constraints von CET liefert uns jetzt die Proposition 4.3 auf Seite 48 ein Entscheidungsverfahren für die Subsumptionsrelation von CET auf Booleschen Kombinationen von Constraints von CET.

Kapitel 7

Quantoreneliminierung für FCET

In diesem Kapitel werden wir den Beweis dafür liefern, daß es sich bei der Theorie FCET, einer Erweiterung der Theorie CET, um eine vollständige Theorie handelt. Der Beweis wird uns zudem noch ein Entscheidungsverfahren für die Gültigkeit beliebiger Formeln über einer endlichen Signatur, die als Relationszeichen nur $\doteq$ enthalten, an die Hand geben.

7.1 Die Theorie FCET

Im letzten Kapitel haben wir das Constraintsystem CET(Σ) kennengelernt. Die Theorie dieses Constraintsystems ist jedoch unvollständig, wenn die Signatur Σ endlich ist. Denn wenn die Signatur endlich ist, so kann, intuitiv gesprochen, ein Term nur mit endlich vielen unterschiedlichen Funktionszeichen anfangen.

Betrachten wir die Formel

$$\forall x \bigvee_{f \in \Sigma} \exists \overline{y}_f \; x \doteq f(\overline{y}_f), \tag{7.1}$$

die wir im weiteren mit DCA(Σ) oder kürzer mit DCA für „domain closure axiom“ bezeichnen werden.

Diese Formel zeigt uns, daß CET(Σ) für endliche Signatur nicht vollständig ist. Denn einerseits gilt für die Struktur $\mathcal{H}$

$$\mathcal{H} \models \mathrm{DCA},$$

andererseits jedoch für die Struktur $\mathcal{T}$ (die volle Termalgebra)

$$\mathcal{T} \not\models \mathrm{DCA},$$

da diese ja auch noch Variablen enthält.

Wenn wir jedoch den Satz DCA(Σ) zur Theorie von CET(Σ) hinzunehmen, erhalten wir eine vollständige Theorie. Diese Theorie bezeichnen wir mit FCET, wobei das „F“ für endlich steht, sie ist also definiert als

$$\mathrm{FCET}(\Sigma) := \mathrm{CET}(\Sigma) \cup \{\mathrm{DCA}(\Sigma)\}.$$

Im weiteren wollen wir gerade beweisen, daß es sich bei FCET(Σ) im Falle, daß Σ endlich ist, um eine vollständige Theorie handelt. Dazu werden wir die Methode der Quantoren-eliminierung verwenden. Dieses Art von Verfahren wurde zuerst von Alfred Tarski [Tar53] für die Axiomatisierung reeller Zahlen verwendet.

Für den Rest dieses Kapitels nehmen wir eine endliche Signatur Σ als fixiert an. Zusätzlich soll diese Signatur es erlauben, unendlich viele Terme zu erzeugen.

Zunächst werden wir einen kleinen Einschub betrachten, indem wir Hilfsmittel bereitstellen, die es uns erleichtern werden, die Terminierung von Ableitungsrelationen einzusehen.

7.2 Wohlfundierte Ordnungen

Wohlfundierte Ordnungen sind ein wichtiges Werkzeug in der Informatik, um die Terminierung von Verfahren einzusehen.

Eine **partielle Ordnung** ist ein Paar $(M, \leq)$, wobei $\leq$ eine binäre Relation auf der Menge M ist, die reflexiv ($\forall m \in M : \; m \leq m$), antisymmetrisch ($\forall m, n \in M : \; m \leq n$ und $n \leq m \Rightarrow n = m$) und transitiv ($\forall m, n, o \in M : \; n \leq m$ und $m \leq o \Rightarrow n \leq o$) ist.

Im folgenden werden wir von einer partiellen Ordnung $(M, \leq)$ oft zu der zugeordneten **strikten Ordnung** $(M, <)$ übergehen (und umgekehrt), wobei für die beiden Relationen $\leq$ und $<$ die folgende Beziehung gilt:

$$n < m \iff n \leq m \text{ und } n \neq m \qquad (\forall n, m \in M).$$

Weiterhin werden wir für $n \leq m$ auch $m \geq n$, und für $n < m$ auch $m > n$ schreiben.

Definition 7.1 (Wohlfundierte Ordnung) *Eine partielle Ordnung $(M, \leq)$ heißt* **wohlfundiert**, *falls es keine unendlich absteigende Folge*

$$m_1 > m_2 > m_3 > \cdots$$

in M gibt.

Proposition 7.1 *Sei $(M, \leq)$ eine partielle Ordnung. Dann sind die folgenden beiden Aussagen äquivalent.*

1. *$(M, \leq)$ ist wohlfundiert.*

2. *Jede Teilmenge $N \subseteq M$ hat ein kleinstes Element.*

Wohlfundierte Ordnungen sind deshalb wichtig, weil sie Induktion der folgenden Form erlauben:

Induktionsanfang Eine Behauptung $\varphi(m)$ gilt für jedes kleinste Element $m \in M$.

Induktionsschritt Falls für alle m mit $m < n$ die Behauptung $\varphi(m)$ gilt, so gilt auch die Behauptung für $\varphi(n)$.

Induktiver Schluß Die Behauptung $\varphi(m)$ gilt für alle $m \in M$.

Beispiel 7.1 (Strukturelle Induktion über Formeln) Eine Ihnen bereits vertraute Schlußweise ist Strukturelle Induktion über den Aufbau von Formeln. Dieses ist jedoch nichts anderes als Induktion über der wohlfundierten Ordnung, die aus der Menge der Formeln und der Teilformel–Relation besteht. □

Im folgenden werden wir ein paar Konstruktionen von wohlfundierten Relationen angeben, die mächtige und oft verwendete Hilfsmittel sind.

7.2.1 Lexikographische Ordnungen

Zwei wichtige Konstruktionen werden wir betrachten.

Definition 7.2 (Lexikographische Ordnung auf Tupeln) *Seien $(M_i, \leq_i)$ wohlfundierte Ordnungen für $i = 1, \ldots, k$. Dann ist die* **lexikographische Ordnung auf Tupeln** *$(M_1 \times \cdots \times M_k, \leq_\times)$ wie folgt definiert:*

$$\begin{aligned} &(m_1, \ldots, m_k) <_\times (n_1, \ldots, n_k) \\ &\quad :\Longleftrightarrow \quad \exists i \in \{1, \ldots, k-1\} : \; m_1 = n_1, \ldots, m_i = n_i \text{ und } m_{i+1} <_{i+1} n_{i+1} \end{aligned}$$

Definition 7.3 (Lexikographische Ordnung auf Wörtern) *Sei $(M, \leq)$ eine wohlfundierte Ordnung. Dann ist die* **lexikographische Ordnung auf Wörtern** *$(M^*, \leq_*)$ wie folgt induktiv definiert:*

$$\begin{aligned} \epsilon <_* mu \quad &:\Longleftrightarrow \quad u \in M^*, m \in M \\ mu <_* nv \quad &:\Longleftrightarrow \quad m < n, \text{ oder } m = n \text{ und } u <_* v \end{aligned}$$

Das es sich bei diesen beiden Ordnungen wirklich um wohlfundierte Ordnungen handelt, sagt die folgende Proposition.

Proposition 7.2

1. *Seien $(M_i, \leq_i)$ wohlfundierte Ordnungen für $i = 1, \ldots, k$. Dann ist die lexikographische Ordnung auf Tupeln $(M_1 \times \cdots \times M_k, \leq_\times)$ wohlfundiert.*

2. *Sei $(M, \leq)$ eine wohlfundierte Ordnung. Dann ist die lexikographische Ordnung auf Wörtern $(M^*, \leq_*)$ wohlfundiert.*

7.2.2 Multimengen–Ordnungen

Eine extrem mächtige und für unsere Zwecke leicht handhabbare wohlfundierte Ordnung werden wir auf Multimengen definieren.

Eine „normale“ Menge M können wir äquivalent auch durch ihre charakteristische Funktion χ_M wie folgt darstellen

$$\chi_M : \begin{cases} M & \to & \{0,1\} \\ m & \mapsto & \left\{\begin{smallmatrix}0\\1\end{smallmatrix}\right\} \text{ falls } \left\{\begin{smallmatrix}m\in M\\ m\notin M\end{smallmatrix}\right\}. \end{cases}$$

Die gleiche Idee werden wir auch für die Definition von Multimengen verwenden, nur lassen wir als Bildbereich einer Funktion, die eine Multimenge darstellt die Menge der natürlichen Zahlen zu.

Sei M eine beliebige nichtleere Menge. Eine **Multimenge über** M ist eine Abbildung $\mu : M \to \mathbb{N}_0$. Ein Element $m \in M$ ist in μ enthalten, kurz $m \in \mu$, falls $\mu(m) > 0$. Die Vereinigung $\mu \cup \nu$, der Durchschnitt $\mu \cap \nu$ und die Differenz $\mu - \nu$ von Multimengen μ und ν über M sind für alle $m \in M$ wie folgt definiert:

$$\begin{aligned} (\mu \cup \nu)(m) &:= \mu(m) + \nu(m) \\ (\mu \cap \nu)(m) &:= \min\{\mu(m), \nu(m)\} \\ (\mu - \nu)(m) &:= \max\{0, \mu(m) - \nu(m)\} \end{aligned}$$

Die leere Multimenge, das heißt die Funktion die überall null ist, bezeichnen wir mit $\emptyset$.

Wir wollen eine Multimenge μ über M **endlich** nennen, wenn die Menge $\{m \in M | \mu(m) > 0\}$ endlich ist. Die Menge der endlichen Multimengen über einer Menge M wollen wir mit $FMS(M)$ bezeichnen. Endliche Multimengen spezifizieren wir manchmal auch durch die Angabe ihrer unter Umständen mehrfach auftretenden Elemente in der Form $\{\!|m_1, \ldots, m_n|\!\}$.

Jeder endlichen Multimenge μ über M, wobei $(M, \leq)$ eine partielle Ordnung ist, können wir ein Wort aus M^* eineindeutig mittels der folgenden, induktiv definierten, Abbildung Ω zuordnen:

$$\begin{aligned} \Omega(\emptyset) &:= \epsilon \\ \Omega(\mu) &:= m\Omega(\mu - \{\!|m|\!\}), m \text{ ist maximales Element in } \mu \text{ bzgl.} \leq \end{aligned}$$

Das heißt wir erhalten $\Omega(\mu)$ durch Hintereinanderschreiben aller Elemente in μ in der durch $\leq$ gegebenen Reihenfolge.

Definition 7.4 (Multimengen–Ordnung) *Sei $(M, \leq)$ eine wohlfundierte Ordnung und $(M^*, \leq_*)$ die zugehörige lexikographische Ordnung auf Wörtern. Dann ist die* **Multimengen–Ordnung** $(FMS(M), \leq_\mu)$ *für alle $\mu, \nu \in FMS(M)$ wie folgt definiert:*

$$\mu \leq_\mu \nu :\Longleftrightarrow \Omega(\mu) \leq_* \Omega(\nu).$$

Proposition 7.3 *Sei $(M, \leq)$ eine wohlfundierte Ordnung. Dann ist die Multimengen–Ordnung $(FMS(M), \leq_\mu)$ wohlfundiert.*

Syntax

$$\sigma,\tau \quad : \quad \Sigma\text{–Formeln}$$

Reduktion

$$\begin{array}{lll}
\neg(\sigma \land \tau) & \longrightarrow & \neg\sigma \lor \neg\tau \\
\neg(\sigma \lor \tau) & \longrightarrow & \neg\sigma \land \neg\tau \\
\neg\neg\sigma & \longrightarrow & \sigma \\
\neg\exists x\, \sigma & \longrightarrow & \forall x\, \neg\sigma \\
\neg\forall x\, \sigma & \longrightarrow & \exists x\, \neg\sigma \\
\neg\top & \longrightarrow & \bot \\
\neg\bot & \longrightarrow & \top
\end{array}$$

Abbildung 7.1: Simplifikationsregeln für Negationsnormalform

Das nützliche an Multimengen–Ordnungen ist, daß man von einer Multimenge durch Herausnehmen eines Elements und durch simultanes Einfügen beliebig vieler echt kleinerer Elemente (bezüglich $<$) ein echt kleineres Element (bezüglich $<_\mu$) erhält.

7.3 Normalformen von Formeln

Um die Präsentation des eigentlichen Verfahrens zur Quantoreneliminierung zu vereinfachen werden wir Formeln oft als in spezieller Normalform gegeben voraussetzen. In diesem Abschnitt werden wir ein paar Verfahren vorstellen, die Formeln in Normalform bringen.

7.3.1 Negationsnormalform

Die erste Normalform von Formeln wird die Eigenschaft haben, nur noch bestimmte Auftreten von Negationszeichen zuzulassen.

Definition 7.5 (Negationsnormalform) *Eine Formel σ ist in* **Negationsnormalform***, wenn alle Negationszeichen in σ direkt vor einer atomaren Formel auftreten.*

In Abbildung 7.1 ist ein Kalkül angegeben, der es erlaubt, aus einer logischen Formel σ eine logische äquivalente Formel in Negationsnormalform darzustellen. Die wesentlichen Aussagen über diesen Kalkül liefert die folgende Proposition.

Proposition 7.4 *Für den Kalkül aus Abbildung 7.1 gelten die folgenden Aussagen.*

Syntax

Formeln	σ, τ	:	Σ–Formeln in Negationsnormalform
Quantoren	Q	::=	$\exists \mid \forall$

Kongruenz

- $\wedge$, $\vee$ kommutativ.
- $Qx\,\sigma \equiv Qy\,\sigma[y/x]$ falls y frei für x in σ.

Reduktion

$$Qx\,\sigma \wedge \tau \longrightarrow Qx\,(\sigma \wedge \tau) \qquad \text{falls } x \notin \mathcal{V}(\tau)$$

$$Qx\,\sigma \vee \tau \longrightarrow Qx\,(\sigma \vee \tau) \qquad \text{falls } x \notin \mathcal{V}(\tau)$$

Abbildung 7.2: Simplifikationsregeln für Pränex–Normalform

1. *$\longrightarrow$ ist terminierend.*
2. *$\longrightarrow \subseteq \models\mid$.*
3. *Sei σ irreduzibel bezüglich $\longrightarrow$. Dann ist σ in Negationsnormalform.*

Beweis Die erste Eigenschaft sieht man ein, wenn man die Multimenge der Höhen der Auftreten von Negationszeichen in einer Formel betrachtet.

Die zweite Eigenschaft ist trivial, die dritte durch einfaches Nachrechnen aller Fälle, wann eine Regel anwendbar ist, einzusehen. □

7.3.2 Pränex–Normalform

Vielfach wird die Untersuchung von Formeln erleichtert, wenn man annehmen kann, daß die Quantoren außen in der Formel stehen, und der Rest der Formel quantorenfrei ist. Dies führt zur

Definition 7.6 (Pränex–Normalform) *Eine Formel σ ist in* **Pränex–Normalform**, *wenn sie die Gestalt*

$$Q_1 x_1 \cdots Q_n x_n\, \tau \qquad (n \geq 0, Q_i \in \{\exists, \forall\})$$

hat, wobei τ quantorenfrei ist. Dabei heißt $Q_1 x_1 \cdots Q_n x_n$ **Quantorenpräfix von** *σ und τ* **Matrix von** *σ.*

In Abbildung 7.2 ist ein Kalkül zur Simplifikation von Formeln in Negationsnormalform gegeben. Das zweite Axiom für die Kongruenz nennt man auch „gebundene Umbennenung“ oder „α–Umbennenung“[1]. Die Eigenschaften dieses Kalküls faßt die nächste Proposition zusammen.

Proposition 7.5 *Für den Kalkül aus Abbildung 7.2 gelten die folgenden Aussagen.*

1. $\longrightarrow$ *ist terminierend.*
2. $(\equiv \cup \longrightarrow) \ \subseteq \models\!\mid$.
3. *Sei σ irreduzibel bezüglich $\longrightarrow$. Dann ist σ in Pränex–Normalform.*

Beweis Interessant ist wiederum nur, wie man die erste Eigenschaft einsehen kann. Als Komplexitätsmaß reicht die Summe der Tiefen der Auftreten von Quantoren in einer Formel. □

Zusammen mit Proposition 7.4 haben wir natürlich auch ein Verfahren zum Herstellen einer Pränex–Normalform zu einer beliebigen Formel. Zuerst stellen wir mit den Regeln aus Abbildung 7.1 Negationsnormalform, dann Pränex–Normalform her.

7.3.3 Disjunktive Normalform

Eine weitere Normalform werden wir nur für quantorenfreie Formeln definieren.

Definition 7.7 (Disjunktive Normalform) *Eine quantorenfreie Formel σ ist in* **disjunktiver Normalform**, *kurz* **DNF**, *falls sie die Form*

$$\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} L_{ij} \qquad (n, m \geq 0)$$

hat, wobei die L_{ij} Literale, das heißt atomare Formeln oder negierte atomare Formeln, sind.

Die Teilformeln $\bigwedge_{j=1}^{m} L_{ij}$ werden wir manchmal als **konjunktive Klauseln** oder einfach nur als **Klauseln** bezeichnen.

Abbildung 7.3 definiert einen Kalkül, dessen irreduzible Formeln in disjunktiver Normalform sind, genauer gilt die folgende Proposition.

Proposition 7.6 *Für den Kalkül aus Abbildung 7.3 gelten die folgenden Aussagen.*

1. $\longrightarrow$ *ist terminierend.*
2. $(\equiv \cup \longrightarrow) \ \subseteq \models\!\mid$.

[1] α–Umbennenung ist die beim λ–Kalkül übliche Bezeichnung für die konsistente Umbennenung von gebundenen Variablen.

Syntax

σ, τ : quantorenfreie Formel in Negationsnormalform

Kongruenz

- $\wedge$, $\vee$ assoziativ und kommutativ

Reduktion

$$(\sigma \vee \sigma') \wedge \tau \quad \longrightarrow \quad (\sigma \wedge \tau) \vee (\sigma' \wedge \tau)$$

Abbildung 7.3: Simplifikationsregeln für Disjunktive Normalform

3. *Sei σ irreduzibel bezüglich $\longrightarrow$. Dann ist σ in disjunktiver Normalform.*

Beweis Wiederum ist wieder nur die erste Aussage interessant. Dazu definieren wir das folgende Komplexitätsmaß $\|\cdot\|$ auf $[\sigma]$ (wobei L ein Literal bezeichnet):

$$\begin{aligned} \|L\| &:= 1 \\ \|\sigma \vee \tau\| &:= \|\sigma\| + \|\tau\| \\ \|\sigma \wedge \tau\| &:= \|\sigma\|^2 \cdot \|\tau\|^2 \end{aligned}$$

Damit gilt für $\sigma \equiv \tau$, daß $\|\sigma\| = \|\tau\|$, da sowohl $+$ als auch $\cdot$ selber wieder assoziativ und kommutativ sind.

Andererseits gilt aber auch

$$\begin{aligned} \|(\sigma \vee \sigma') \wedge \tau\| &= \|\sigma \vee \sigma'\|^2 \cdot \|\tau\|^2 \\ &= (\|\sigma\| + \|\sigma'\|)^2 \cdot \|\tau\|^2 \\ &= \left(\|\sigma\|^2 + 2 \cdot \|\sigma\| \cdot \|\sigma'\| + \|\sigma'\|^2\right) \cdot \|\tau\|^2 \\ &> \left(\|\sigma\|^2 + \|\sigma'\|^2\right) \cdot \|\tau\|^2 \\ &= \|\sigma\|^2 \cdot \|\tau\|^2 + \|\sigma'\|^2 \cdot \|\tau\|^2 \\ &= \|\sigma \wedge \tau\| + \|\sigma' \wedge \tau\| \\ &= \|(\sigma \wedge \tau) \wedge (\sigma' \wedge \tau)\|, \end{aligned}$$

das heißt $\longrightarrow$ ist terminierend. □

Wiederum haben wir nicht nur ein Verfahren das quantorenfreie Formeln in Negationsnormalform zu einer äquivalenten Form in DNF simplifiziert, sondern zusammen mit Proposition 7.4 haben wir auch ein Verfahren, daß beliebige quantorenfreie Formeln in DNF umformt.

Analog zu der disjunktiven Normalform können wir auch die **konjunktive Normalform**, kurz **KNF**, definieren und herstellen. Damit ist eine quantorenfreie Formel in konjunktiver Normalform, wenn sie die Gestalt

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} L_{ij} \qquad (n, m \geq 0)$$

hat, wobei die L_{ij} wiederum Literale sind.

Erhalten können wir sie, wenn wir einen Kalkül analog zu dem in Abbildung 7.3 definieren, der die Reduktionsregel

$$(\sigma \wedge \sigma') \vee \tau \longrightarrow (\sigma \vee \tau) \wedge (\sigma' \vee \tau)$$

hat.

7.4 Ein Quantoreneliminierungsschema

Im folgenden werden wir Gebrauch von den folgenden logischen Äquivalenzen machen, die in allen Strukturen gültig sind.

$$\begin{array}{rcll}
\exists x\,(\sigma \vee \tau) & \models\!\mid & \exists x\,\sigma \vee \exists x\,\tau & \quad (7.2)\\
\forall x\,(\sigma \wedge \tau) & \models\!\mid & \forall x\,\sigma \wedge \forall x\,\tau & \quad (7.3)\\
Qx\,\sigma & \models\!\mid & \sigma \qquad \text{falls } x \notin \mathcal{V}(\sigma), Q \in \{\exists, \forall\} & \quad (7.4)\\
\neg\sigma \vee \tau & \models\!\mid & \neg\sigma \vee (\sigma \wedge \tau) & \quad (7.5)\\
\exists x(x \doteq s \wedge \sigma) & \models\!\mid & \sigma[s/x] \qquad \text{falls } x \notin \mathcal{V}(s) & \quad (7.6)
\end{array}$$

Zusätzlich führen wir noch die folgenden Konventionen für den Rest dieses Kapitels ein: M und N bezeichnen quantorenfreie Formeln, und K bezeichnet Konjunktionen von Literalen. Für die Formel $\neg s \doteq t$ schreiben wür auch kürzer $s \not\doteq t$.

Als Begriff werden wir verwenden: τ heißt **Simplifikation von σ bezüglich Δ**, oder kürzer $\sigma \overset{\Delta}{\rightsquigarrow} \tau$, falls

$$\sigma \models\!\mid_{\Delta} \tau \qquad \text{und} \qquad \mathcal{V}(\tau) \subseteq \mathcal{V}(\sigma).$$

Satz 7.7 *Sei ein Simplifikationsverfahren*

$$\exists \overline{x}\, K \overset{\Delta}{\rightsquigarrow} \forall \overline{y}\, M \tag{7.7}$$

gegeben. Dann existiert ein Verfahren

$$\sigma \overset{\Delta}{\rightsquigarrow} \forall \overline{y}\, M.$$

Beweis Wir werden ein induktives Verfahren angeben, daß selbst die folgenden Grundschritte zur Eliminierung von Quantorenalternierungen, das heißt Wechseln in der Art der Quantoren, benutzt.

1. $\boxed{\exists\overline{x}\, M \overset{\Delta}{\rightsquigarrow} \forall\overline{y}\, N}$ (7.8)
   Dieses Verfahren ergibt sich wie folgt:

$$
\begin{array}{rlll}
\exists\overline{x}\, M & \models\!\!\mid & \exists\overline{x} \bigvee_{i=1}^{n} K_i & \left(\bigvee_{i=1}^{n} K_i \text{ ist DNF von } M\right) \\
& \models\!\!\mid & \bigvee_{i=1}^{n} \exists\overline{x}\, K_i & (\text{wegen } (7.2)) \\
& \models\!\!\mid_\Delta & \bigvee_{i=1}^{n} \forall\overline{y}_i\, M_i & (\text{wegen } (7.7)\ , \mathcal{V}(\overline{y}_i) \cap \mathcal{V}(\overline{y}_j) = \emptyset \text{ für } i \neq j) \\
& \models\!\!\mid & \forall\overline{y} \bigvee_{i=1}^{n} M_i & (\overline{y} = \overline{y}_1 \cdots \overline{y}_n)
\end{array}
$$

   Wir setzen also N zu $\bigvee_{i=1}^{n} M_i$.

2. $\boxed{\forall\overline{x}\exists\overline{y}\, M \overset{\Delta}{\rightsquigarrow} \forall\overline{z}\, N}$ (7.9)
   Mit (7.8) simplifizieren wir $\exists\overline{y}\, M$ direkt zu $\forall\overline{z}'\, N$ und erhalten wie gewünscht die Formel $\forall\overline{x}\forall\overline{z}'\, N = \forall\overline{z}\, N$.

3. $\boxed{\exists\overline{x}\forall\overline{y}\, M \overset{\Delta}{\rightsquigarrow} \forall\overline{z}\, N}$ (7.10)
   Das Simplifikationsverfahren (7.8) liefert im wesentlichen auch dieses Verfahren:

$$
\begin{array}{rlll}
\exists\overline{x}\forall\overline{y}\, M & \models\!\!\mid & \exists\overline{x}\, \neg\exists\overline{y}\, \neg M & \\
& \models\!\!\mid_\Delta & \exists\overline{x}\, \neg\forall\overline{y}'\, M' & (\text{mit Verfahren } (7.8)) \\
& \models\!\!\mid & \exists\overline{x}\exists\overline{y}'\, \neg M' & \\
& \models\!\!\mid & \forall\overline{y}\, N & (\text{mit Verfahren } (7.8))
\end{array}
$$

Zuerst wird σ auf die Pränex–Normalform $Q_1\overline{x}_1 Q_2\overline{x}_2 \cdots Q_k\overline{x}_k\, M$ gebracht, wobei $Q_i \in \{\exists, \forall\}$ für $i = 1, \ldots, k$ und $Q_i \neq Q_{i+1}$ für $i = 1, \ldots, k-1$. Nun führen wir das eigentliche Verfahren mit Induktion über die Anzahl der Quantorenalternierungen k durch. Dazu müssen wir die folgenden Fälle unterscheiden.

1. $\boxed{k = 0, \text{ oder } k = 1 \text{ und } Q_1 = \forall}$
   Damit sind wir fertig.
2. $\boxed{k = 1 \text{ und } Q_1 = \exists}$
   Wir wenden das Verfahren (7.8) an.
3. $\boxed{k > 1 \text{ und } Q_{k-1}\overline{x}_{k-1}Q_k\overline{x}_k = \exists\overline{x}_{k-1}\forall\overline{x}_k}$
   Zuerst wenden wird das Verfahren (7.9) auf $\exists\overline{x}_{k-1}\forall\overline{x}_k\, M$ an, erhalten somit insgesamt eine Formel $Q_1\overline{x}_1 \cdots Q_{k-2}\overline{x}_{k-2}\forall\overline{z}\, N$. Auf diese wenden wir das Verfahren wieder rekursiv an.
4. $\boxed{k > 1 \text{ und } Q_{k-1}\overline{x}_{k-1}Q_k\overline{x}_k = \forall\overline{x}_{k-1}\exists\overline{x}_k}$
   Analog zu dem vorigen Fall mit Verwendung von (7.10).

□

Satz 7.8 *Seien die Verfahren*

$$\exists \overline{x}\, K \overset{\Delta}{\rightsquigarrow} \forall \overline{y}\, M \tag{7.11}$$

und

$$\exists\, K \overset{\Delta}{\rightsquigarrow} \left\{ {\bot \atop \top} \right\} \tag{7.12}$$

gegeben. Dann gibt es ein Verfahren

$$\forall\, \sigma \overset{\Delta}{\rightsquigarrow} \left\{ {\bot \atop \top} \right\}.$$

Beweis Wegen Satz 7.7 genügt es, ein Verfahren

$$\forall\, M \overset{\Delta}{\rightsquigarrow} \left\{ {\bot \atop \top} \right\}$$

anzugeben.

$$\begin{array}{rcll} \forall\, M & \models\!\!\mid & \neg\exists\neg M & \\ & \models\!\!\mid & \neg\exists \bigvee\limits_{i=1}^{n} K_i & \left(\bigvee\limits_{i=1}^{n} K_i \text{ DNF von } \neg M\right) \\ & \models\!\!\mid & \neg \bigvee\limits_{i=1}^{n} \exists K_i & \\ & \models\!\!\mid & \neg \bigvee\limits_{i=1}^{n} L_i & (\text{mit Verfahren (7.12)}, L_i \in \{\bot, \top\}) \end{array}$$

Nun ist es trivial, die Disjunktion mit Literalen $\bot$ oder $\top$ nach $\left\{ {\bot \atop \top} \right\}$ zu simplifizieren. Geleistet wird dies zum Beispiel von den Regeln

$$\bot \vee \sigma \longrightarrow \sigma \qquad \text{und} \qquad \top \vee \sigma \longrightarrow \top$$

modulo Assoziativität und Kommutativität von $\vee$. □

7.5 Simplifikation von Quasiklauseln

Jetzt sind wir in der Lage, die in den Sätzen 7.7 und 7.8 angegebenen Verfahren auf die Theorie FCET anzuwenden. Dafür müßen wir nur die zwei verlangten Verfahren, einmal (7.7) (beziehungsweise (7.11)) und (7.12) für FCET angeben.

In Abbildung 7.4 sind Syntax und Kongruenz für den Kalkül angegeben, der uns die beiden Verfahren liefern wird. Der Ausdruck $\langle Q\rangle$ ist einfach als Q zu lesen, die Einklammerung dient nur dem Zweck, um eine eindeutige Zerlegung von ω–Ausdrücken in Quasiklauseln zu gewährleisten, da sowohl Ausdrücke als auch Quasiklauseln durch $\wedge$ verbunden werden können.

Bevor wir uns mit dem zweiten Teil der Eliminierung von $\exists$–Quantoren, der Ableitungsrelation, beschäftigen, definieren wir Komplexitätsmaße auf Elementen von $[Q]$ und $[\omega]$.

Einer Quasiklausel Q ordnen wir das Tripel $\|Q\|_Q := (n_1, n_2, n_3)$ als Komplexität zu, wobei die Komponenten dieses Tripels wie folgt definiert sind:

Syntax

Constraints:	φ, ψ	$::=$	$s \doteq t \quad \mid \quad \bot \quad \mid \quad \top \quad \mid \quad \varphi \wedge \psi$
Quasiklauseln:	Q	$::=$	$\varphi \quad \mid \quad \neg\varphi \quad \mid \quad Q \wedge Q' \quad \mid \quad \exists x\, Q$
Ausdrücke:	ω	$::=$	$\langle Q \rangle \quad \mid \quad \omega \wedge \omega' \quad \mid \quad \omega \vee \omega' \quad \mid \quad \forall x\, \omega$

Kongruenz

$\wedge$: assoziativ, kommutativ und $\omega \wedge \top \equiv \omega$.

$\doteq$: $s \doteq t \equiv t \doteq s$

$\exists$: α–Umbennenung, $\exists x \exists y\, Q \equiv \exists y \exists x\, Q$ und $\exists x\, Q \wedge Q' \equiv \exists x(Q \wedge Q')$, falls $x \notin \mathcal{V}(Q')$.

Abbildung 7.4: Eliminierung von $\exists$–Quantoren, Teil (a)

1. n_1 ist gleich der Anzahl der $\exists$–Quantoren in Q.
2. n_2 ist die Summe der Tiefen der Auftreten der existentiell quantifizierten Variablen in den positiven Gleichungen in Q. Zum Beispiel ist die Tiefe des Auftretens von X in der Formel $\exists \mathsf{X}\, \mathsf{Y} \doteq \mathsf{f}(\mathsf{X})$ gleich 2, und in der Formel $\exists \mathsf{X}\, \mathsf{Y} \doteq \mathsf{f}(\mathsf{f}(\mathsf{X}))$ gleich 3.
3. n_3 ist die Anzahl der Auftreten von Funktionssymbole und Variablen in Q.

Einem Ausdruck ω ordnen wir als Komplexität die Multimenge $\|\omega\|_\omega$ der Komplexitäten aller durch $\langle\cdot\rangle$–geklammerten Quasiklauseln zu, also

$$\|\omega\|_\omega := \{\!|\, \|Q\|_Q \mid \langle Q \rangle \text{ in } \omega |\!\}.$$

Wir werden sehen, daß dieses Komplexitätsmaß es uns erlauben wird, die Terminierung der noch anzugebenden Reduktionsregeln zu beweisen.

In Abbildung 7.5 sind die Reduktionsregeln angegeben, die uns die gewünschten Verfahren für FCET liefern werden. Die für uns wichtigen Eigenschaften der Reduktionsrelation sind in der folgenden Proposition zusammengefaßt.

Proposition 7.9

1. *Die Regeln (7.13) bis (7.19) sind Äquivalenztransformationen für* CET, *insbesondere also auch für* FCET, *das heißt*

$$\longrightarrow \subseteq \models\!|_{\mathrm{CET}} \subseteq \models\!|_{\mathrm{FCET}} .$$

2. *Die Relation $\longrightarrow$ ist terminierend.*

Beweis

Reduktion

$x \doteq x$	$\longrightarrow$	$\top$		(7.13)
$f(\overline{s}) \doteq f(\overline{t})$	$\longrightarrow$	$\overline{s} \doteq \overline{t}$		(7.14)
$f(\overline{s}) \doteq g(\overline{t})$	$\longrightarrow$	$\bot$	falls $f \neq g$	(7.15)
$x \doteq f(\overline{s})$	$\longrightarrow$	$\bot$	falls $x \in \mathcal{V}(\overline{s})$	(7.16)
$\exists x\,(x \doteq s \wedge Q)$	$\longrightarrow$	$Q[s/x]$	falls $x \notin \mathcal{V}(s)$	(7.17)
$\exists x \left(\bigwedge_{i=1}^{n} y_i \not\doteq s_i\right)$	$\longrightarrow$	$\top$		(7.18)

$n \geq 0$, falls für alle $i = 1, \ldots, n$: $y_i \notin \mathcal{V}(s_i)$ und $x \in \mathcal{V}(y_i) \cup \mathcal{V}(s_i)$

$\langle \exists \overline{x}\,(\neg(\varphi \wedge \psi) \wedge Q)\rangle$	$\longrightarrow$	$\langle \exists \overline{x}\,(\neg\varphi \wedge Q)\rangle \vee \langle \exists \overline{x}\,(\neg\psi \wedge Q)\rangle$	falls $\varphi, \psi \not\equiv \top$	(7.19)

Abbildung 7.5: Eliminierung von $\exists$–Quantoren, Teil (b)

1. Bekannt ist bereits, daß die Regeln (7.13) bis (7.16) Äquivalenztransformationen in CET sind. Weiterhin ist (7.17) gerade (7.6).

   Für $n = 0$ ist $\bigwedge_{i=1}^{n} y_i \not\doteq s_i$ gleich $\top$, in diesem Fall ist (7.18) trivialerweise eine Äquivalenztransformation. Für den Fall $n > 0$ sagt dieses gerade Malc'evs Lemma 6.15 aus (Man beachte im Beweis zu diesem Lemma insbesondere (6.12)).

   Zu untersuchen bleibt noch (7.19), also:

$$
\begin{aligned}
\exists \overline{x}\,(\neg(\varphi \wedge \psi) \wedge Q) &\models\!\mid \exists \overline{x}\,((\neg\varphi \vee \neg\psi) \wedge Q) \\
&\models\!\mid \exists \overline{x}\,((\neg\varphi \wedge Q) \vee (\neg\psi \wedge Q)) \\
&\models\!\mid \exists \overline{x}\,(\neg\varphi \wedge Q) \vee \exists \overline{x}\,(\neg\psi \wedge Q) \qquad \text{(wegen (7.2))}
\end{aligned}
$$

2. Um die Terminierung des Verfahrens einzusehen, überzeugen wir uns zuerst, daß aus $\omega \equiv \omega'$ auch $\|\omega\|_\omega = \|\omega'\|_\omega$ folgt. Dies liegt einerseits daran, daß Assoziativität von $\wedge$ und $\vee$ nichts am Komplexitätsmaß verändern, und daß $\|\langle\top\rangle\|_\omega = \{\!|(0,0,0)|\!\}$ gilt. Andererseits verändern die Regeln für $\exists$ weder die Anzahl der Existenzquantoren, noch die Tiefe des Auftretens der existentiell quantifizierten Variablen.

   Für die Regeln (7.13) bis (7.16) bleibt die Anzahl der Existenzquantoren gleich, die Tiefe der Auftreten von existentiell quantifizierten Variablen steigt nicht an, und die Anzahl der Funktionszeichen und Variablen nimmt echt ab, also sinkt das Komplexitätsmaß $\|\cdot\|_Q$.

   Für die Regeln (7.17) und (7.18) nimmt jeweils die Anzahl der Existenzquantoren ab, das heißt die erste Komponente von $\|\cdot\|_Q$ nimmt echt ab.

Bei Anwendung der Regel (7.19) bleibt die Anzahl der Existenzquantoren gleich, die Tiefe der Auftreten existentiell quantifizierter Variablen steigt nicht an, aber auf jeden Fall nimmt die Anzahl der Funktionszeichen und Variablen echt ab, da sowohl φ als auch ψ nicht kongruent zu $\top$ sind. Da $\|\cdot\|_\omega$ eine Multimengen–Ordnung ist, nimmt diese echt ab.

$\square$

Die nächste Proposition sagt aus, daß uns die Reduktionsregeln bereits das Verfahren

$$\exists K \overset{\text{CET}}{\rightsquigarrow} \left\{ \begin{smallmatrix} \perp \\ \top \end{smallmatrix} \right\}$$

liefern. Aber leider können noch nicht alle $\exists$–Quantoren eliminiert werden, zum Beispiel:

$$\exists \mathsf{X}\, \mathsf{Y} \doteq \mathsf{f}(\mathsf{X}) \qquad \text{ist irreduzibel!}$$

Zumindest werden wir angeben, wie irreduzible Formeln genau aussehen.

Proposition 7.10 *Sei Q irreduzibel bezüglich der Regeln aus Abbildung 7.5. Dann gilt:*

1. *Falls Q einen Existenzquantor enthält, so gilt:*
$$Q \equiv \exists \overline{x}\, (y \doteq f(\overline{s}) \wedge Q')\,. \tag{7.20}$$

2. *Falls Q geschlossen ist, ist Q eine Boolesche Kombination von $\top$ und $\perp$.*

Beweis

1. Nun werden wir Schritt für Schritt durchspielen, daß Q wirklich von der angegebenen Form ist.
   - Q enthält nach Voraussetzung mindestens einen Existenzquantor.
   - Jede Gleichung $t \doteq t'$ läßt sich mit Hilfe der Regeln (7.13) bis (7.16) in die Form $y \doteq f(\overline{s})$ bringen.
   - y kann nicht in $\mathcal{V}(\overline{x})$ enthalten sein, da sonst die Reduktionsregel (7.17) anwendbar wäre.
   - y kann aber auch nicht in $\mathcal{V}(\overline{s})$ enthalten sein, da sonst die Gleichung $y \doteq f(\overline{s})$ mit der Regel (7.16) zu $\perp$ simplifizieren würde.
   - Außerdem muß $y \notin \mathcal{V}(\overline{x}) \cap \mathcal{V}(\overline{s})$ gelten, da sonst die Quantoren $\exists x$ mit $x \notin \mathcal{V}(\overline{s})$ mit der Kongruenz $\equiv$ in Q'' aufträten.
   - Zudem können nicht nur Ungleichungen auftreten, da diese durch die Regel (7.18) simplifizierbar wären.
2. Da Q geschlossen ist, kann eine irreduzible Form von der Gestalt (7.20) nicht existieren, da dann die Bedingung $y \notin \mathcal{V}(\overline{x}) \cup \mathcal{V}(\overline{s})$ verletzt wäre. Mit dieser Beobachtung und der obigen Argumentation folgt die Behauptung.

$\square$

Es ist zu beachten, daß wir nicht wie gefordert nach $\top$ oder $\bot$ sondern zu einer Booleschen Kombination von diesen simplifizieren. Aber im Beweis zu Satz 7.7 haben wir bereits gesehen, wie die weitere Simplifikation zu funktionieren hat.

Die nächste Proposition liefert uns das noch fehlende Stück für die vollständige Simplifikation von existentiell quantifizierten Quasiklauseln.

Proposition 7.11 (Explosion) *In* FCET(Σ) *gilt die Äquivalenz*

$$\exists\overline{x}\,(y \doteq f(\overline{s}) \wedge Q)$$

$$\models\!\!|_{\mathrm{FCET}(\Sigma)}$$

$$\bigwedge_{\substack{g\in\Sigma\\ g\neq f}} \forall\overline{z}_g\; y \not\doteq g(\overline{z}_g) \wedge \forall\overline{z}\,(y \not\doteq f(\overline{z}) \vee y \doteq f(\overline{z}) \wedge \exists\overline{x}\,(\overline{z} \doteq \overline{s} \wedge Q))$$

falls $y \notin \mathcal{V}(\overline{x})$ *und die universell quantifizierten Variablen* $\overline{z}_g$ *und* $\overline{z}$ *in* $\exists\overline{x}(y \doteq f(\overline{s}) \wedge Q)$ *nicht vorkommen.*

Zudem ist

$$\|\langle\exists\overline{x}\,(y \doteq f(\overline{s}) \wedge Q)\rangle\|_\omega >_\mu$$
$$\left\|\bigwedge_{\substack{g\in\Sigma\\ g\neq f}} \langle\forall\overline{z}_g\; y \not\doteq g(\overline{z}_g)\rangle \wedge \forall\overline{z}\,(\langle y \not\doteq f(\overline{z})\rangle \vee \langle y \doteq f(\overline{z}) \wedge \exists\overline{x}\,(\overline{z} \doteq \overline{s} \wedge Q)\rangle)\right\|_\omega$$

Beweis Sei $\tau := \exists\overline{x}\,(y \doteq f(\overline{s}) \wedge Q)$. Dann gilt

$$\begin{array}{lll}
\tau & \models\!\!| & \neg\top \vee \tau \\
 & \models\!\!|_{\mathrm{FCET}} & \neg \bigvee_{g\in\Sigma} \exists\overline{z}_g\; y \doteq g(\overline{z}_g) \vee \tau \qquad (\mathrm{DCA}(\Sigma)) \\
 & \models\!\!| & \bigwedge_{g\in\Sigma} \forall\overline{z}_g\; y \not\doteq g(\overline{z}_g) \vee \tau \\
 & \models\!\!| & \bigwedge_{g\in\Sigma} \forall\overline{z}_g\,(y \not\doteq g(\overline{z}_g) \vee \tau) \\
 & \models\!\!| & \bigwedge_{g\in\Sigma} \forall\overline{z}_g\,(y \not\doteq g(\overline{z}_g) \vee (y \doteq g(\overline{z}_g) \wedge \tau)) \qquad (\text{wegen } (7.5)) \\
 & \models\!\!|_{\mathrm{CET}} & \bigwedge_{\substack{g\in\Sigma\\ g\neq f}} \forall\overline{z}_g\; y \not\doteq g(\overline{z}_g) \wedge \forall\overline{z}\,(y \not\doteq f(\overline{z}) \vee y \doteq f(\overline{z}) \wedge \tau) \\
 & \models\!\!|_{\mathrm{CET}} & \bigwedge_{\substack{g\in\Sigma\\ g\neq f}} \forall\overline{z}_g\; y \not\doteq g(\overline{z}_g) \wedge \forall\overline{z} \begin{pmatrix} y \not\doteq f(\overline{z}) \vee \\ y \doteq f(\overline{z}) \wedge \exists\overline{x}\,(\overline{z} \doteq \overline{s} \wedge Q) \end{pmatrix}
\end{array}$$

Daß das Komplexitätsmaß $\|\cdot\|_\omega$ echt sinkt, liegt daran, daß in den ersten beiden Quasiklauseln $\langle\cdot\rangle$ überhaupt keine Existenzquantoren mehr vorkommen. Das Komplexitätsmaß $\|\cdot\|_Q$ für die dritte Quasiklausel sinkt deshalb, weil die Tiefe der existentiell quantifizierten Variablen $\overline{x}$, das heißt die zweite Komponente des Komplexitätsmaßes sinkt, und die Anzahl der Existenzquantoren, das heißt die erste Komponente, konstant bleibt. $\square$

Wenn wir also den Regeln aus Abbildung 7.5 noch die Regel

$$\langle \exists \overline{x}\,(y \doteq f(\overline{s}) \wedge Q)\rangle \longrightarrow \bigwedge_{\substack{g\in\Sigma\\ g\neq f}} \langle \forall \overline{z}_g\; y \not\doteq g(\overline{z}_g)\rangle \wedge \forall \overline{z}\,(\langle y \not\doteq f(\overline{z})\rangle \vee \langle y \doteq f(\overline{z}) \wedge \exists \overline{x}\,(\overline{z} \doteq \overline{s} \wedge Q)\rangle)$$

hinzufügen, erhalten wir das Simplifizierungsverfahren

$$\exists \overline{x}\, K \overset{\text{FCET}}{\leadsto} \forall \overline{y}\, M,$$

wie es die Sätze 7.7 und 7.8 in (7.7), respektive (7.8) verlangen. Diese Regel wird dann auch **Explosionsregel** genannt.

Proposition 7.12 *Sei ω irreduzibel bezüglich der Reduktionsregeln aus Abbildung 7.5 und der Explosionsregel. Dann enthält ω keinen Existenzquantor.*

Beweis Nehmen wir an, daß ω bezüglich der angegebenen Regeln irreduzibel ist und noch einen Existenzquantor enthält.

Die Quasiklausel, die den Existenzquantor enthält, kann aber nach Proposition 7.10 nicht geschlossen sein. Also muß sie die in Proposition 7.10 unter 1. angegebene Gestalt besitzen. Das kann aber nicht sein, da dann noch die Explosionsregel anwendbar wäre! □

Damit erhaben wir das angestrebte Ziel erreicht, es bleibt nur noch der folgende Satz zu formulieren.

Satz 7.13 (Vollständigkeit von FCET) *Die Theorie* FCET(Σ) *ist vollständig.*

Beweis Für jeden prädikatenlogischen Satz σ können wir entscheiden, daß

$$\sigma \models\!\mid_{\text{FCET}} \left\{ \begin{smallmatrix} \bot \\ \top \end{smallmatrix} \right\}$$

gilt, mit anderen Worten entweder gilt FCET $\models \sigma$ oder FCET $\models \neg\sigma$. □

Die Vollständigkeit der Theorie FCET wurde erstmalig von Kunen 1987 [Kun87] im Zusammenhang mit „Negation als endlichem Scheitern“ vermutet, jedoch nicht bewiesen. Maher zeigte dann 1988 [Mah88], daß sowohl FCET als auch CET für unendliche Signatur vollständige Theorien sind. Gleichzeitig haben Comon und Lescanne [CL88] unabhängig von Maher gezeigt, daß FCET vollständig ist; daran lehnt sich unsere Darstellung im wesentlichen an. Später stellte sich heraus, daß die Vollständigkeit von FCET leicht aus einem viel früheren Resultat von Mal’cev [Mal71] folgt.

Kapitel 8

Effiziente Unifikation und rationale Bäume

Wir haben bereits in Kapitel 6 ein Verfahren zur Unifikation vorgestellt. Ein großer Nachteil dieses Verfahrens ist, daß es sogar exponentielle Laufzeit beanspruchen kann. In diesem Kapitel werden wir ein Verfahren vorstellen, dessen Laufzeit quasi–linear ist.

8.1 Auffalten von Termen

Die Idee bei der effizienten Unifikation wird sein, nur Gleichungen zwischen einer Variable und einem **flachen Term** zuzulassen, das heißt ein Term mit maximal einem Funktionszeichen. Durch eine geschickte Manipulation dieser flachen Terme werden wir das Kopieren von Termen verhindern können, damit verhindern wir das (eventuell exponentielle) Anwachsen der Terme.

Die nächsten beiden Propositionen sind Hilfsmittel für die Transformation von Termen in die oben erwähnte flache Darstellung.

Proposition 8.1 (Auffaltung) *Sei $x \notin \mathcal{V}(s)$ und sei s frei für x in σ. Dann gilt:*

$$\sigma[s/x] \models\mid \exists x\,(x \doteq s \wedge \sigma)\,.$$

Proposition 8.2 *Eine Formel $\exists\overline{x}\,\sigma$ ist erfüllbar in einer Struktur $\mathcal{A}$ genau dann, wenn σ in $\mathcal{A}$ erfüllbar ist.*

Das eigentliche Herstellen der flachen Repräsentation beschreibt die folgende Proposition.

Proposition 8.3 *Sei σ eine Formel gemäß*

$$\sigma ::= s \doteq t \quad | \quad \sigma \wedge \sigma' \quad | \quad \exists\sigma$$

Dann kann man in linearer Zeit eine zu σ $\models\mid$-äquivalente Formel τ berechnen, gemäß

$$\begin{array}{rcl} u & ::= & x \quad | \quad f(x_1,\ldots,x_n) \\ \tau & ::= & x \doteq u \quad | \quad \tau \wedge \tau' \quad | \quad \exists x\,\tau \end{array}$$

Das nächste Beispiel gibt die Transformation einer Gleichung in eine flache Darstellung wieder.

Beispiel 8.1 Sei die folgende Gleichung gegeben:

$$\mathsf{f(X)} \doteq \mathsf{g(h(a, f(X)))}.$$

Eine logisch äquivalente Formel ist die folgende:

$$\exists \mathsf{Y} \exists \mathsf{Z} \exists \mathsf{U} \exists \mathsf{V} \left[\begin{array}{l} \mathsf{Y} \doteq \mathsf{f(X)} \wedge \mathsf{Y} \doteq \mathsf{g(Z)} \\ \mathsf{Z} \doteq \mathsf{h(U,V)} \wedge \mathsf{U} \doteq \mathsf{a} \wedge \mathsf{V} \doteq \mathsf{f(X)} \end{array} \right],$$

wobei diese Darstellung gerade der in Proposition 8.3 geforderten entspricht. Wie im weiteren, steht die Matrix gerade für die Konjunktion ihrere Zeilen. □

8.2 Das Simplifikationssystem COL

Jetzt werden wir ein Simplifikationssystem vorstellen, daß gerade auf den in Proposition 8.3 eingeführten Ausdrücken rechnet.

In Abbildung 8.1 ist das Simplifikationssystem COL angegeben. Die Regel (8.4) wird auch **Konfrontationsregel** genannt. Das Simplifikationssystem ist nach Alain Colmerauer, einem der Miterfinder der Programmiersprache PROLOG, benannt; von ihm wurde dieses System erstmals in [Col84] vorgestellt.

Proposition 8.4 *Für alle σ existieren $\overline{x}$ und φ mit:*

$$\sigma \equiv \exists \overline{x}\, \varphi.$$

Beweis Die Argumentation verläuft wie beim Herstellen der Pränex–Normalform, siehe dazu Abschnitt 7.3.2 auf Seite 72. □

Im weiteren wollen wir mit CET_0 die Theorie bezeichen, die nur Instantiierungen der ersten beiden Axiomenschemata von CET umfaßt, also

$$\mathrm{CET}_0 : \left\{ \begin{array}{lll} (\mathrm{Ax1}) & \forall\, (f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n) \rightarrow \bigwedge_{i=1}^{n} x_i \doteq y_i) & \\ (\mathrm{Ax2}) & \forall\, (f(x_1, \ldots, x_n) \doteq g(y_1, \ldots, y_n) \rightarrow \bot) & (f \neq g) \end{array} \right.$$

Proposition 8.5 *Für das Simplifikationssystem* COL *gelten die folgenden Aussagen:*

- $\sigma \equiv \tau \;\Rightarrow\; \sigma \models\!\mid \tau.$
- $\sigma \longrightarrow \tau \;\Rightarrow\; \sigma \models\!\mid_{\mathrm{CET}_0} \tau.$

Bevor wir das System COL weiter untersuchen, werden wir ein Beispiel betrachten.

Syntax

$$
\begin{array}{lcl}
u, v & ::= & x \quad | \quad f(x_1, \ldots, x_n) \\
\varphi, \psi & ::= & x \doteq u \quad | \quad \bot \quad | \quad \top \quad | \quad \varphi \wedge \psi \\
\sigma, \tau & ::= & \varphi \quad | \quad \exists x\, \sigma \quad | \quad \sigma \wedge \tau
\end{array}
$$

Kongruenz

$\wedge$: assoziativ, kommutativ, $\varphi \wedge \top \equiv \varphi$

$\exists$: α-Umbenennung, $\exists x \exists y\, \sigma \equiv \exists y \exists x\, \sigma$, $\exists x\, \sigma \wedge \tau \equiv \exists x(\sigma \wedge \tau)$ falls $x \notin \mathcal{V}(\sigma)$

Reduktion

$$x \doteq x \wedge \varphi \longrightarrow \varphi \tag{8.1}$$

$$x \doteq y \wedge \varphi \longrightarrow x \doteq y \wedge \varphi[y/x] \qquad (y \neq x, x \in \mathcal{V}(\varphi)) \tag{8.2}$$

$$\bot \wedge \varphi \longrightarrow \bot \qquad (\varphi \not\equiv \top) \tag{8.3}$$

$$x \doteq f(\overline{y}) \wedge x \doteq f(\overline{z}) \wedge \varphi \longrightarrow x \doteq f(\overline{y}) \wedge \overline{y} \doteq \overline{z} \wedge \varphi \tag{8.4}$$

$$x \doteq f(\overline{y}) \wedge x \doteq g(\overline{z}) \wedge \varphi \longrightarrow \bot \wedge \varphi \qquad (f \neq g) \tag{8.5}$$

$$\frac{\sigma \longrightarrow \tau}{\exists x\, \sigma \longrightarrow \exists x\, \tau} \tag{8.6}$$

$$\exists x(x \doteq y \wedge \varphi) \longrightarrow \varphi[y/x] \qquad (x \neq y) \tag{8.7}$$

$$\exists y(x \doteq y \wedge \varphi) \longrightarrow \varphi[x/y] \qquad (x \neq y) \tag{8.8}$$

$$\exists x\, \varphi \longrightarrow \varphi \qquad (x \notin \mathcal{V}(\varphi)) \tag{8.9}$$

Abbildung 8.1: Das Simplifikationssystem COL

Beispiel 8.2 Sei die Formel

$$\mathsf{X} \doteq \mathsf{Y} \wedge \mathsf{X} \doteq \mathsf{f}(\mathsf{f}(\mathsf{X})) \wedge \mathsf{Y} \doteq \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{Y})))$$

gegeben. Um diese mit dem System COL zu simplifizieren überführen wir sie zunächst mit Proposition 8.3 in die logisch äquivalente Formel

$$\exists \mathsf{Z}_1 \exists \mathsf{Z}_2 \exists \mathsf{Z}_3\, (\mathsf{X} \doteq \mathsf{Y} \wedge \mathsf{X} \doteq \mathsf{f}(\mathsf{Z}_1) \wedge \mathsf{Z}_1 \doteq \mathsf{f}(\mathsf{X}) \wedge \mathsf{Y} \doteq \mathsf{f}(\mathsf{Z}_2) \wedge \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Z}_3) \wedge \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Y}))\,.$$

Zunächst simplifizieren wir die Matrix dieser Formel. Dabei markieren wir den Teil der Formel der bei der Simplifikation relevant ist jeweils durch das Einklammern mit $\langle\!\langle\cdot\rangle\!\rangle$.

$$\begin{bmatrix} \langle\!\langle\, \mathsf{X} \doteq \mathsf{Y} \,\rangle\!\rangle \\ \mathsf{X} \doteq \mathsf{f}(\mathsf{Z}_1) \\ \mathsf{Z}_1 \doteq \mathsf{f}(\mathsf{X}) \\ \mathsf{Y} \doteq \mathsf{f}(\mathsf{Z}_2) \\ \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Z}_3) \\ \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Y}) \end{bmatrix} \xrightarrow{(8.2)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Y} \\ \langle\!\langle\, \mathsf{Y} \doteq \mathsf{f}(\mathsf{Z}_1) \,\rangle\!\rangle \\ \mathsf{Z}_1 \doteq \mathsf{f}(\mathsf{Y}) \\ \langle\!\langle\, \mathsf{Y} \doteq \mathsf{f}(\mathsf{Z}_2) \,\rangle\!\rangle \\ \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Z}_3) \\ \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Y}) \end{bmatrix} \xrightarrow{(8.4)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Y} \\ \mathsf{Y} \doteq \mathsf{f}(\mathsf{Z}_1) \\ \mathsf{Z}_1 \doteq \mathsf{f}(\mathsf{Y}) \\ \langle\!\langle\, \mathsf{Z}_1 \doteq \mathsf{Z}_2 \,\rangle\!\rangle \\ \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Z}_3) \\ \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Y}) \end{bmatrix}$$

$$\xrightarrow{(8.2)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Y} \\ \mathsf{Y} \doteq \mathsf{f}(\mathsf{Z}_2) \\ \langle\!\langle\, \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Y}) \,\rangle\!\rangle \\ \mathsf{Z}_1 \doteq \mathsf{Z}_2 \\ \langle\!\langle\, \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Z}_3) \,\rangle\!\rangle \\ \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Y}) \end{bmatrix} \xrightarrow{(8.4)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Y} \\ \mathsf{Y} \doteq \mathsf{f}(\mathsf{Z}_2) \\ \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Y}) \\ \mathsf{Z}_1 \doteq \mathsf{Z}_2 \\ \langle\!\langle\, \mathsf{Y} \doteq \mathsf{Z}_3 \,\rangle\!\rangle \\ \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Y}) \end{bmatrix}$$

$$\xrightarrow{(8.2)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Z}_3 \\ \langle\!\langle\, \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Z}_2) \,\rangle\!\rangle \\ \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Z}_3) \\ \mathsf{Z}_1 \doteq \mathsf{Z}_2 \\ \mathsf{Y} \doteq \mathsf{Z}_3 \\ \langle\!\langle\, \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Z}_3) \,\rangle\!\rangle \end{bmatrix} \xrightarrow{(8.4)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Z}_3 \\ \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Z}_2) \\ \mathsf{Z}_2 \doteq \mathsf{f}(\mathsf{Z}_3) \\ \mathsf{Z}_1 \doteq \mathsf{Z}_2 \\ \mathsf{Y} \doteq \mathsf{Z}_3 \\ \langle\!\langle\, \mathsf{Z}_2 \doteq \mathsf{Z}_3 \,\rangle\!\rangle \end{bmatrix}$$

$$\xrightarrow{(8.2)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Z}_3 \\ \langle\!\langle\, \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Z}_3) \,\rangle\!\rangle \\ \langle\!\langle\, \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Z}_3) \,\rangle\!\rangle \\ \mathsf{Z}_1 \doteq \mathsf{Z}_3 \\ \mathsf{Y} \doteq \mathsf{Z}_3 \\ \mathsf{Z}_2 \doteq \mathsf{Z}_3 \end{bmatrix} \xrightarrow{(8.4)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Z}_3 \\ \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Z}_3) \\ \langle\!\langle\, \mathsf{Z}_3 \doteq \mathsf{Z}_3 \,\rangle\!\rangle \\ \mathsf{Z}_1 \doteq \mathsf{Z}_3 \\ \mathsf{Y} \doteq \mathsf{Z}_3 \\ \mathsf{Z}_2 \doteq \mathsf{Z}_3 \end{bmatrix}$$

$$\xrightarrow{(8.1)} \begin{bmatrix} \mathsf{X} \doteq \mathsf{Z}_3 \\ \mathsf{Z}_3 \doteq \mathsf{f}(\mathsf{Z}_3) \\ \mathsf{Z}_1 \doteq \mathsf{Z}_3 \\ \mathsf{Y} \doteq \mathsf{Z}_3 \\ \mathsf{Z}_2 \doteq \mathsf{Z}_3 \end{bmatrix}$$

Jetzt erhalten wir durch dreimaliges Anwenden von (8.6) auf jeden der vorherigen Schritte:

$$\exists Z_1 \exists Z_2 \exists Z_3 \begin{bmatrix} X \doteq Y \\ X \doteq f(Z_1) \\ Z_1 \doteq f(X) \\ Y \doteq f(Z_2) \\ Z_2 \doteq f(Z_3) \\ Z_3 \doteq f(Y) \end{bmatrix} \longrightarrow^* \exists Z_1 \exists Z_2 \exists Z_3 \begin{bmatrix} X \doteq Z_3 \\ Z_3 \doteq f(Z_3) \\ Z_1 \doteq Z_3 \\ Y \doteq Z_3 \\ Z_2 \doteq Z_3 \end{bmatrix}$$

Nun können wir weiter simplifizieren, mittels:

$$\begin{aligned} \exists Z_1 \exists Z_2 \exists Z_3 \begin{bmatrix} X & \doteq & Z_3 \\ Z_3 & \doteq & f(Z_3) \\ Z_1 & \doteq & Z_3 \\ Y & \doteq & Z_3 \\ Z_2 & \doteq & Z_3 \end{bmatrix} & \overset{(8.8)}{\longrightarrow} \exists Z_1 \exists Z_2 \begin{bmatrix} X & \doteq & Y \\ Y & \doteq & f(Y) \\ Z_1 & \doteq & Y \\ Z_2 & \doteq & Y \end{bmatrix} \\ & \overset{(8.7)}{\longrightarrow} \exists Z_1 \begin{bmatrix} X & \doteq & Y \\ Y & \doteq & f(Y) \\ Z_1 & \doteq & Y \end{bmatrix} \\ & \overset{(8.7)}{\longrightarrow} \begin{bmatrix} X & \doteq & Y \\ Y & \doteq & f(Y) \end{bmatrix} \end{aligned}$$

Damit sind wir fertig, insgesamt haben wir also:

$$X \doteq Y \wedge X \doteq f(f(X)) \wedge Y \doteq f(f(f(Y))) \models\!\mid_{\mathrm{CET}_0} X \doteq Y \wedge Y \doteq f(Y).$$

□

Wir nennen eine Variable x **eliminiert in** σ genau dann, wenn $\overline{x}$, y und φ existieren, so daß

$$\sigma \equiv \exists \overline{x} \, (x \doteq y \wedge \varphi) \, ,$$

wobei $x \neq y$ und $x \notin \mathcal{V}(\varphi)$. Man beachte bei dieser Definition, daß es nach Proposition 8.4 eine Zerlegung in Quantorenpräfix und Matrix gibt.

Proposition 8.6 *Die Reduktionsrelation* $\longrightarrow$ *von* COL *ist terminierend.*

Beweis Wir definieren das Komplexitätsmaß $\|\sigma\|$ auf $[\sigma]$ als Quadrupel mit denen folgenden Komponenten in der angegebenen Reihenfolge:

1. Anzahl der Existenzquantoren in σ.
2. Anzahl der nicht eliminierten Variablen in σ.
3. Anzahl der Auftreten von Funktionssymbolen in σ.
4. Anzahl der Auftreten von Variablen in σ.

Es ist leicht einzusehen, daß die Kongruenz $\equiv$ dieses Komplexitätsmaß invariant läßt.

Für die Regeln (8.1) bis (8.5) geben wir jeweils nur an, welche Komponente des Komplexitätsmaßes auf jeden Fall herabgesetzt wird: daß die Komponenten höherer Priorität zumindest nicht ansteigen, ist leicht einzusehen.

Regel (8.1) senkt die Anzahl der Variablen, Regel (8.2) erhöht die Anzahl der eliminierten Variablen, Regel (8.3) senkt die Anzahl der Variablen oder die Anzahl der Funktionszeichen und die beiden Regeln (8.4) und (8.5) die Anzahl der Funktionszeichen.

Auch (8.6) ist unkritisch: Denn wenn $\|\sigma\| >_\times \|\tau\|$ gilt, folgt direkt $\|\exists x\, \sigma\| >_\times \|\exists x\, \tau\|$.

Die Anzahl der Existenzquantoren wird in den letzten drei Regeln (8.7) bis (8.9) jeweils um eins herabgesetzt. □

Eine besondere Art von Substitution ist die in der folgenden Definition eingeführte.

Definition 8.1 (Normierer) *Ein* **Normierer** *ist eine endliche idempotente Substitution, die Variablen auf Variablen abbildet.*

Im weitern werden wir die Konvention benutzen, mit θ sowohl den Normierer

$$[x_1/y_1, \ldots, x_n/y_n]$$

als auch das zugehörige Gleichungssystem

$$x_1 \doteq y_1 \wedge \cdots \wedge x_n \doteq y_n$$

zu bezeichnen. Dabei ist zu beachten, daß in dem Gleichungssystem insbesondere auch $x_i \neq y_i$ gilt.

Definition 8.2 (Determinante) *Eine* **Determinante** δ *ist eine Formel*

$$\begin{array}{rcl} x_1 & \doteq & f_1(\overline{y}_1) \wedge \\ & \vdots & \\ x_n & \doteq & f_n(\overline{y}_n) \end{array}$$

mit $n \geq 0$ *und* $x_1, \ldots, x_n$ *paarweise verschieden. Für eine Determinante* δ *definieren wir den* **Bereich von** δ *als*

$$\mathcal{D}(\delta) := \{x_1, \ldots, x_n\}.$$

Eine Determinante δ heißt **azyklisch**, wenn sie kongruent ist (bezüglich $\equiv$) zu einer Determinante

$$\begin{array}{rcl} x_1 & \doteq & f_1(\overline{y}_1) \wedge \\ & \vdots & \\ x_n & \doteq & f_n(\overline{y}_n), \end{array}$$

so daß für alle $i = 1, \ldots, n$:

$$x_i \notin \mathcal{V}(\overline{y}_i) \cup \cdots \cup \mathcal{V}(\overline{y}_n).$$

Eine Formel $\theta \wedge \exists \overline{x}\, \delta$ heißt **kanonisch**, wenn die folgenden Bedingungen erfüllt sind:

- θ ist Normierer und δ Determinante.
- $\mathcal{D}om(\theta) \parallel \mathcal{V}(\delta)$, $\mathcal{V}(\theta) \parallel \mathcal{V}(\overline{x})$ und $\mathcal{V}(\overline{x}) \subseteq \mathcal{V}(\delta)$.

Proposition 8.7 *Sei σ bezüglich der Reduktionsrelation von* COL *irreduzibel. Dann ist $\sigma = \perp$ oder σ ist kongruent zu einer kanonischen Formel.*

Beweis Wir setzen $\sigma \neq \perp$ voraus. Nach Proposition 8.4 können wir annehmen, daß σ kongruent zu

$$\exists \overline{x}\, (\theta \wedge \delta) \tag{8.10}$$

ist, wobei θ nur Gleichungen der Form $x \doteq y$ und δ nur Gleichungen der Form $x \doteq f(\overline{y})$ enthält (Die Fälle, wo θ oder δ kongruent zu $\top$ sind, sind sowieso trivial).

Wir zeigen nun die folgenden Schritte:

θ ist ein Normierer: Trivialerweise ist θ endlich. Zusätzlich enthält θ keine Gleichung $x \doteq x$, da sonst Regel (8.1) anwendbar wäre, außerdem auch keine zwei Gleichungen $x \doteq y$ und $x \doteq z$, da sonst die Regel (8.2) anwendbar wäre. Also können wir θ direkt mit der zugehörigen Substitution identifizieren.

Wir nehmen an, daß θ nicht idempotent sei. Dann gibt es ein $x \in \mathcal{D}om(\theta)$ mit $(\theta \circ \theta)(x) \neq \theta(x)$, das heißt $\theta(x) = y$ und $\theta(y) = z$, wobei x, y und z paarweise verschieden sind. $y \neq z$ gilt. Sei o.B.d.A. (die restlichen drei Fälle sind symmetrisch) $\theta \equiv y \doteq x \wedge y \doteq z \wedge \cdots$. Damit ist aber Regel (8.2) anwendbar, da $y \neq x$ und $y \in \mathcal{V}(y \doteq z)$, also haben wir einen Widerspruch!

δ ist eine Determinante: Sei $\delta \equiv x_1 \doteq f_1(\overline{y}_1) \wedge \cdots \wedge x_n \doteq f_n(\overline{y}_n)$. Nehmen wir an, daß es $i, j \in \{1, \ldots, b\}$ mit $x_i = x_j$ und $i \neq j$ gibt. Wenn jetzt $f_i = f_j$ gälte, so wäre Regel (8.4) anwendbar. Also ist $f_i \neq f_j$. Das führt zu einem Widerspruch, da sonst die Regel (8.5) anwendbar ist!

$\mathcal{D}om(\theta) \parallel \mathcal{V}(\delta)$, $\mathcal{V}(\theta) \parallel \mathcal{V}(\overline{x})$ und $\mathcal{V}(\overline{x}) \subseteq \mathcal{V}(\delta)$: Wäre $\mathcal{D}om(\theta) \cap \mathcal{V}(\delta) \neq \emptyset$, so wäre wiederum Regel (8.2) anwendbar.

Auch muß $\mathcal{V}(\theta) \parallel \mathcal{V}(\overline{x})$ gelten, denn sonst wäre eine der Regeln (8.7) oder (8.8) anwendbar.

Gäbe es andererseits ein x mit $x \in \mathcal{V}(\overline{x})$ und $x \notin \mathcal{V}(\delta)$, so wäre, da $\mathcal{V}(\theta) \parallel \mathcal{V}(\overline{x})$ gilt, die Regel (8.9) anwendbar.

Da, wie bereits gezeigt, $\mathcal{V}(\theta) \parallel \mathcal{V}(\overline{x})$ gilt, ist σ kongruent zu der kanonischen Formel

$$\theta \wedge \exists \overline{x}\, \delta.$$

$\square$

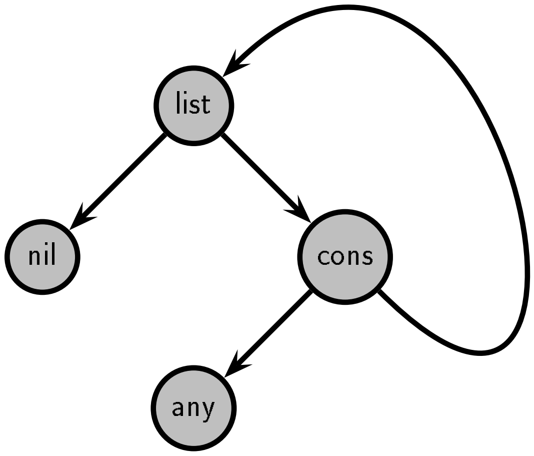


Abbildung 8.2: Typbeschreibung für Listen

Proposition 8.8 *Sei $\sigma = \theta \wedge \exists \overline{x}\ \delta$ eine kanonische Formel. Dann sind die folgenden Aussagen zueinander äquivalent.*

1. $\exists \sigma \models\!\mid \top$.
2. $\exists \sigma \models\!\mid_{\mathrm{CET}} \top$.
3. δ *azyklisch.*

Beweis Übungsaufgabe! □

8.3 Die Theorie RAT

Nach dem Herstellen einer Normalform bezüglich des Simplifikationssystems COL müßen wir, um die Erfüllbarkeit zu testen einen Test auf Azyklizität durchführen. Was ist jedoch wenn wir diesen Test auslassen?

Einerseits erreichen wir natürlich dadurch mehr Effizienz, da der Test nicht mehr ausgeführt werden muß. Andererseits werden durch zyklische Normalformen bezüglich COL in endlicher Art und Weise zyklische, also unendliche „Datenstrukturen“ beschrieben. Das dieses kein „Bug“, sondern ein „Feature“ ist, und mehr Expressivität mit sich bringt, ist klar, dazu betrachten wir ein kleines Beispiel.

Beispiel 8.3 Wir wollen ein Typspezifikation für Listen angeben. Umgangssprachlich haben wir: „Eine Liste ist entweder nil oder besteht aus einem mit cons markierten Paar aus irgendeinem Datum und einer Liste.“ Intuitiv entspricht diese Spezifikation dem Baum in Abbildung 8.2, oder der Lösung der Gleichung

$$\mathsf{X} \doteq \mathsf{list}(\mathsf{nil}, \mathsf{cons}(\mathsf{any}, \mathsf{X})).$$

□

In welchem Sinne dadurch Datenstrukturen beschrieben werden, welche Eigenschaften diese haben, und wie damit gerechnet werden kann, wollen wir im weiteren formalisieren und näher untersuchen.

Sei Σ eine Signatur. Die **Theorie der rationalen Bäume** $\mathrm{RAT}(\Sigma)$ ist als die Menge der Formeln definiert, die sich mit Hilfe der folgenden Axiomenschemata bilden lassen.

(Ax1)	$\forall\,(f(x_1,\ldots,x_n) \doteq f(y_1,\ldots,y_n) \rightarrow \bigwedge_{i=1}^n x_i \doteq y_i)$	
(Ax2)	$\forall\,(f(x_1,\ldots,x_n) \doteq g(y_1,\ldots,y_n) \rightarrow \bot)$	$(f \neq g)$
(Ax3)	$\forall\exists\overline{x}\,(\overline{x} \doteq \overline{s})$	($\overline{x} \doteq \overline{s}$ Determinante)
(Ax4)	$\forall\,(\overline{x} \doteq \overline{s} \wedge \overline{y} \doteq \overline{s}[\overline{y}/\overline{x}] \rightarrow \overline{x} \doteq \overline{y})$ ($\overline{x} \doteq \overline{s}$ und $\overline{y} \doteq \overline{s}[\overline{y}/\overline{x}]$ Determinanten)	

Alle Instanzen der ersten drei Axiomenschemata wollen wir als $\mathrm{RAT}_0(\Sigma)$ bezeichen. Das dritte Axiomenschemata spezifiziert die Lösbarkeit von Determinanten, das vierte die Eindeutigkeit der Lösungen von Determinanten.

Proposition 8.9 *Wenn δ eine Determinante ist, dann gilt:*

$$\mathrm{RAT}_0 \models \exists\delta.$$

Beweis Folgt aus (Ax3), da $\sigma \models \exists x\,\sigma$. □

Bis jetzt haben wir nur ausgesagt, daß jede Determinante in allen Modellen von RAT_0 erfüllbar ist. Um zu zeigen, daß RAT_0 konsistent ist, müssen wir zeigen, daß es ein Modell von RAT_0 gibt.

8.4 Baumbereiche und Bäume

In diesem Abschnitt werden wir uns mit unendlichen Bäumen beschäftigen. Als Gerüst (in zweifacher Hinsicht) werden wir Mengen betrachten, die unmarkierte Bäume beschreiben, sogenannte Baumbereiche.

Eine nichtleere Menge $B \subseteq M^*$ heißt **Baumbereich über der Menge** M, wenn B präfixabgeschlossen ist, das heißt jeder Präfix eines Wortes aus B ist wiederum in B enthalten, oder formal: falls $u \in B$ und $u = vw$, so ist $v \in B$. Insbesondere enthält also jeder Baumbereich das leere Wort ϵ.

Ein Baumbereich $B \subseteq M^*$ heißt **regulär** oder **rational**, wenn es eine endliche Teilmenge F von M gibt, mit $B \subseteq F^*$ und B ist eine reguläre Sprache über F.

Beispiel 8.4 In Abbildung 8.5 ist ein unendlicher Baumbereich skizziert, der auch durch den regulären Ausdruck

$$(\mathsf{bc})^*\,(\mathsf{a} + \mathsf{b}\,(\epsilon + \mathsf{a} + \mathsf{b}))$$

gegeben ist. □

Bekannte Charakterisierungen für eine reguläre Sprache B sind die folgenden:

Geordneter Baum

Ordnung durch Kantenmarkierungen

Abbildung 8.3: Geordnete versus markierte Bäume

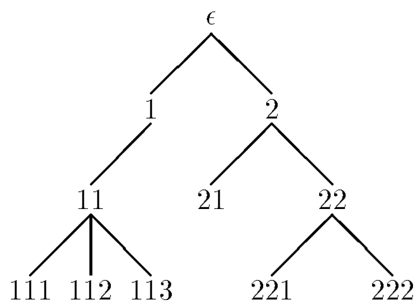


Abbildung 8.4: Skizze eines Baumbereichs

- der minimale endliche deterministische Automat für B ist eine bis auf Isomorphie der Zustände eindeutige festgelegte, endliche Darstellung für B.

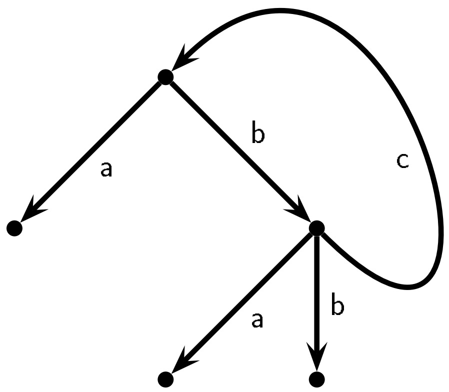


Abbildung 8.5: Darstellung eines unendlichen Baums durch einen Graphen

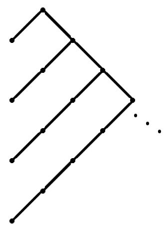


Abbildung 8.6: Ein nichtrationaler Baum

- B hat nur endlich viele verschiedene Unterbäume. Oder in Ausdrücken der Automatentheorie nach dem Satz von Myhill–Nerode [HU79]: B ist Vereinigung von Äquivalenzklassen einer rechtsinvarianten Äquivalenzrelation von endlichem Index[1].

Ein nichtrationaler Baumbereich ist in Abbildung 8.6 zu sehen.

Sei Σ im folgenden eine prädikatenlogische Signatur, die nur Funktionssymbole enthält. Wir werden uns jetzt dem zweiten Schritt in der Konstruktion unendlicher Bäume zuwenden, indem wir Elemente eines Baumbereichs mit einer konsistenten Markierung versehen.

Ein Σ–**Baum** ist eine Abbildung

$$T : B \to \Sigma,$$

so daß B ein Baumbereich über $\mathbb{N}$ ist und für alle $u \in B$ gilt:

$$\{k \in \mathbb{N} \mid uk \in B\} = \{1, \ldots, n\},$$

wobei n die Stelligkeit des Funktionssymbols $T(u) \in \Sigma$ ist.

Die endlichen Σ–Bäume entsprechen gerade den bisher mengentheoretisch definierten geschlossenen Termen über Σ.

Wir nennen einen Σ–Baum $T : B \to \Sigma$ **rational**, falls die Sprache $T(B)$ regulär ist. Dazu äquivalent ist die Aussage, daß T nur endlich viele verschiedene Teilbäume enthält.

Beispiel 8.5 In Abbildung 8.7 ist der Term $\mathsf{f}(\mathsf{f}(\mathsf{b}), \mathsf{h}(\mathsf{c}, \mathsf{a}, \mathsf{b}))$ als Baum dargestellt. Dieser Baum wird durch die Abbildung

$$\{(\epsilon, \mathsf{f}), (1, \mathsf{f}), (11, \mathsf{b}), (2, \mathsf{h}), (21, \mathsf{c}), (22, \mathsf{a}), (23, \mathsf{b})\}$$

beschrieben. □

Die so definierten Bäume geben uns gerade die Universa für verschiedene Σ–Algebren, die für uns von Interesse sind.

[1] Eine Äquivalenzrelation $\sim$ auf Wörtern ist rechtsinvariant, falls aus $u \sim v$ auch $uw \sim vw$ für alle w folgt. Eine Äquivalenzrelation ist von endlichem Index, wenn sie nur endlich viele Äquivalenzklassen hat.

Mit $\mathcal{I}(\Sigma)$ bezeichnen wir die **Algebra der unendlichen Bäume**. Das Universum von $\mathcal{I}(\Sigma)$ ist gerade die Menge aller Σ–Bäume, und jedes Funktionssymbol $f \in \Sigma$ wird wie folgt interpretiert:

$$f^{\mathcal{I}(\Sigma)}(T_1, \ldots, T_n) := \{(\epsilon, f)\} \cup \bigcup_{i=1}^{n} \{(iu, g) \mid (u, g) \in T_i\}.$$

Eine Teilmenge des Universums einer Σ–Algebra beschreibt genau dann eine Subalgebra, also wiederum eine Σ–Algebra, wenn diese Teilmenge unter Anwendung der durch die Signatur gegebenen Funktionen abgeschlossen ist.

Die **Algebra der rationalen Bäume** $\mathcal{R}(\Sigma)$ ist die Subalgebra von $\mathcal{I}(\Sigma)$, deren Universum gerade die rationalen Bäume sind.

Zwei Σ–Strukturen $\mathcal{A}$ und $\mathcal{B}$ sind zueinander **isomorph**, wenn es eine bijektive Abbildung $\iota : \mathcal{U}(\mathcal{A}) \to \mathcal{U}(\mathcal{B})$ gibt, so daß für alle n-stelligen Funktionssymbole $f \in \Sigma$, n–stelligen Prädikatensymbole $p \in \sigma$ und Werte $u_1, \ldots, u_n \in \mathcal{U}(\mathcal{A})$ gilt:

$$\iota\left(f^{\mathcal{A}}(u_1, \ldots, u_n)\right) = f^{\mathcal{B}}(\iota(u_1), \ldots, \iota(u_n))$$

und

$$(u_1, \ldots, u_n) \in p^{\mathcal{A}} \iff (\iota(u_1), \ldots, \iota(u_n)) \in p^{\mathcal{B}}.$$

Für die Subalgebra von $\mathcal{I}(\Sigma)$, die nur endliche Bäume enthält gelten gerade die folgenden Aussagen:

Proposition 8.10 *Die Signatur Σ enthalte mindestens eine Konstante, und $\mathcal{A}$ sei die Subalgebra von $\mathcal{I}(\Sigma)$, deren Universum nur endliche Bäume enthält. Dann gilt:*

1. *$\mathcal{A}$ ist isomorph zu $\mathcal{H}(\Sigma)$.*

2. *$\mathcal{A}$ ist die kleinste Subalgebra von $\mathcal{I}(\Sigma)$.*

Beweis Übungsaufgabe! $\square$

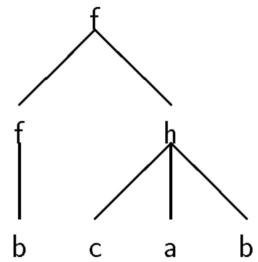


Abbildung 8.7: Ein endlicher Baum

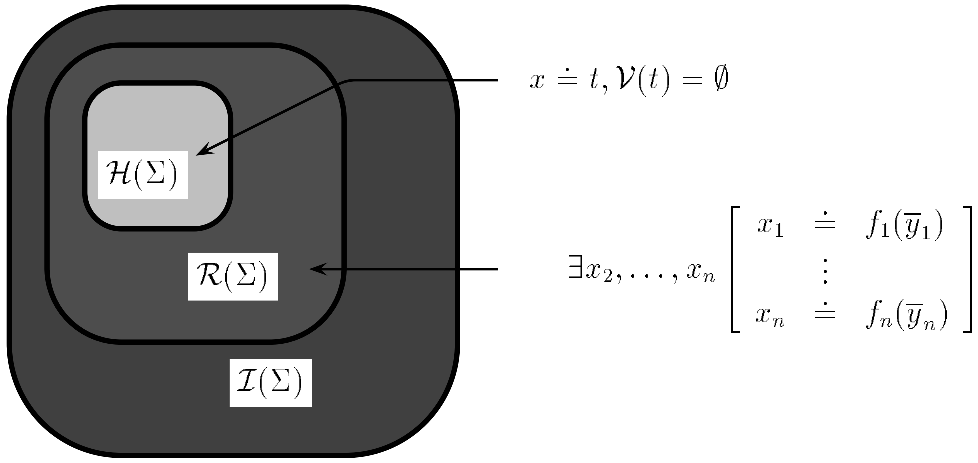


Abbildung 8.8: Die Algebren $\mathcal{I}(\Sigma)$, $\mathcal{R}(\Sigma)$ und $\mathcal{H}(\Sigma)$

Auf Grund der ersten Aussage der obigen Proposition werden wir keinen Unterschied zwischen $\mathcal{A}$ und $\mathcal{H}(\Sigma)$ mehr machen.

Abgeschlossen wird dieser Abschnitt durch die Aussage, daß unsere Konstruktionen genau das liefern, wonach wir gesucht haben.

Satz 8.11 *$\mathcal{I}(\Sigma)$ und $\mathcal{R}(\Sigma)$ sind Modelle von* RAT(Σ)*:*

$$\mathcal{I}(\Sigma) \models \mathrm{RAT} \qquad \textit{und} \qquad \mathcal{R}(\Sigma) \models \mathrm{RAT}.$$

Beweis Übungsaufgabe! □

Die Beziehung zwischen den Algebren $\mathcal{I}(\Sigma)$, $\mathcal{R}(\Sigma)$ und $\mathcal{H}(\Sigma)$, und den Formeln, die Elemente dieser Algebren eindeutig beschreiben, ist in Abbildung 8.8 dargestellt. Dabei läßt sich jedes Element von $\mathcal{H}(\Sigma)$ und $\mathcal{R}(\Sigma)$ durch Gleichungen der angegebenen Form eindeutig beschreiben.

8.5 Das Constraintsystem RAT(Σ)

Im folgenden werden wir das Constraintsystem RAT betrachten. Die Syntax der Constraints ist die, die wir auch schon bei dem Simplifikationssystem COL (siehe Abbildung 8.1) verwendet haben.

Das Constraintsystem RAT(Σ) ist das Tripel $(\Sigma, \mathrm{RAT}(\Sigma), [\sigma])$ mit $[\sigma]$ wie folgt:

$$\begin{array}{rcl} u, v & ::= & x \quad | \quad f(x_1, \ldots x_n) \\ \varphi, \psi & ::= & x \doteq u \quad | \quad \bot \quad | \quad \top \quad | \quad \varphi \wedge \psi \\ \sigma, \tau & ::= & \varphi \quad | \quad \sigma \wedge \tau \quad | \quad \exists x\, \sigma \end{array}$$

Im folgenden lassen wir wieder die Signatur Σ weg. Bisher wissen wir schon die folgende Eigenschaften von RAT: Das Simplifikationssystem COL simplifiziert jeden Constraint in $[\sigma]$ entweder zu $\perp$ oder zu einem kanonischen Constraint (Mittels Äquivalenztransformationen in $\mathrm{CET}_0 \subseteq \mathrm{RAT}_0 \subseteq \mathrm{RAT}$).

Weiterhin gilt die folgende Proposition.

Proposition 8.12 *Wenn σ ein kanonischer Constraint ist, dann*

$$\mathrm{RAT}_0 \models \exists\sigma.$$

Beweis Sei $\sigma = \theta \wedge \exists\overline{x}\,\delta$, dann ist leicht einzusehen:

$$\begin{array}{rcl} \exists(\theta \wedge \exists\overline{x}\,\delta) & \models\!\!| & \exists\exists\overline{x}\,\delta \\ & \models\!\!| & \exists\exists\mathcal{D}(\delta)\,\delta \\ & \models\!\!|_{\mathrm{RAT}_0} & \exists\top \\ & \models\!\!| & \top \end{array}$$

□

Damit können wir also die Erfüllbarkeit eines Constraints in RAT_0 und RAT entscheiden. Mehr noch, sowohl RAT_0 als auch RAT sind erfüllbarkeitsvollständig (Das sagt gerade die Kombination von Proposition 8.12 und Satz 8.11 aus).

Oder anders ausgedrückt: COL entscheidet die Erfüllbarkeit von Constraints in $\mathcal{R}$ und $\mathcal{I}$.

Im weiteren Verlauf werden wir die folgenden logischen Äquivalenzen verwenden.

Proposition 8.13

1. *Seien σ und τ beliebige Formeln, und $\mathcal{V}(\overline{x}) \parallel \mathcal{V}(\tau)$. Dann gilt:*

$$\exists\overline{x}\,\sigma \models \tau \iff \sigma \models \tau. \tag{8.11}$$

2. *Seien σ und τ beliebige Formeln, und θ ein Normierer, so daß, für alle $x \in \mathcal{D}om(\theta)$, $\theta(x)$ frei für x in σ und τ ist. Dann gilt:*

$$\theta \wedge \sigma \models \tau \iff \theta\sigma \models \theta\tau. \tag{8.12}$$

und

$$\theta \wedge \sigma \models\!\!| \theta \wedge \theta\sigma. \tag{8.13}$$

Beweis Durch einfaches Nachrechnen. □

8.6 Kongruenzen und Normierer

Wir beginnen sofort mit einem zentralen Begriff für die Beschreibung der Information, die ein Constraint enthält. Dabei schreiben wir $s \doteq t \in \varphi$, wenn $s \doteq t$ in φ vorkommt.

Definition 8.3 *Eine Äquivalenzrelation auf Variablen ρ ist eine* **Kongruenz von** φ, *falls ρ eine Äquivalenzrelation auf Variablen ist, und die beiden folgenden Eigenschaften erfüllt:*

$$\frac{x \doteq y \in \varphi}{x \; \rho \; y} \qquad\qquad \frac{x \doteq f(\overline{y}) \in \varphi \quad x' \doteq f(\overline{y}') \in \varphi \quad x \; \rho \; x'}{\overline{y} \; \rho \; \overline{y}'}$$

Intuitiv gesehen beschreibt eine Kongruenz eines Constraints mindestens die Gleichheiten zwischen Variablen in einem Constraint, die auf Grund der Injektivität der Konstruktoren implizit in einem Constraint enthalten ist.

Proposition 8.14 *Jeder Constraint φ hat eine kleinste Kongruenz.*

Beweis Man überzeugt sich leicht davon, daß die totale Relation $VAR \times VAR$ eine Kongruenz von φ ist, und daß Kongruenzen von φ unter Durchschnitt erhalten bleiben. □

Beispiel 8.6 Es gilt

$$\left[\begin{array}{lcl} \mathsf{X} & \doteq & \mathsf{f}(\mathsf{Y},\mathsf{X}) \\ \mathsf{Y} & \doteq & \mathsf{f}(\mathsf{Z},\mathsf{Y}) \\ \mathsf{Z} & \doteq & \mathsf{f}(\mathsf{X},\mathsf{X}) \end{array}\right] \models_{\mathrm{RAT}} \mathsf{X} \doteq \mathsf{Y},$$

denn wie man leicht einsieht, beschreiben die Lösungen für X und Y jeweils den vollen binären Baum, der überall mit dem Funktionszeichen f markiert ist.

Die kleinste Kongruenz ρ von

$$\left[\begin{array}{lcl} \mathsf{X} & \doteq & \mathsf{f}(\mathsf{Y},\mathsf{X}) \\ \mathsf{Y} & \doteq & \mathsf{f}(\mathsf{Z},\mathsf{Y}) \\ \mathsf{Z} & \doteq & \mathsf{f}(\mathsf{X},\mathsf{X}) \\ \mathsf{X} & \doteq & \mathsf{Y} \end{array}\right]$$

ist gegeben durch:

$$\rho(\mathsf{X}) = \{\mathsf{X},\mathsf{Y},\mathsf{Z}\}.$$

□

Wir werden Äquivalenzrelationen auf Variablen in der folgenden Art und Weise mit Normierern identifizieren.

Definition 8.4 *Eine Substitution θ heißt genau dann* **Normierer für eine Äquivalenzrelation** ρ *auf Variablen, wenn:*

- *θ Normierer.*

- $x \,\rho\, y \iff \theta(x) = \theta(y)$.

Wir nennen θ einen **Normierer für den Constraint** φ, falls θ Normierer der kleinsten Kongruenz von φ ist.

Im folgenden geben wir ein paar einfache Eigenschaften von Normierern an.

Proposition 8.15 *Wenn $\theta \wedge \delta$ kanonisch ist, so ist θ ein Normierer für $\theta \wedge \delta$.*

Proposition 8.16 *Sei θ ein Normierer. Dann gilt:*

$$\forall x, y \in VAR: \quad \theta(x) = \theta(y) \iff \theta \models x \doteq y.$$

Beweis Übungsaufgabe! □

Proposition 8.17 *Seien θ und θ' Normierer für dieselbe Äquivalenzrelation. Dann gilt:*

$$\theta \models\!\mid \theta'.$$

Ein Constraint φ enthält einen **Konflikt**, wenn $\bot \in \varphi$ oder es gibt $x \doteq f(\overline{y}) \in \varphi$ und $x \doteq g(\overline{z}) \in \varphi$ mit $f \neq g$.

Wir werden im folgenden mit $\overline{\varphi}$ den Constraint bezeichnen, den wir erhalten, wenn wir aus φ alle trivialen Gleichungen $x \doteq x$ und alle Duplikate streichen. Zum Beispiel gilt

$$\overline{\mathsf{Y} \doteq \mathsf{Y} \wedge \mathsf{X} \doteq \mathsf{f(X)} \wedge \mathsf{X} \doteq \mathsf{f(X)}} \quad = \quad \mathsf{X} \doteq \mathsf{f(X)}.$$

Trivialerweise gilt dann natürlich die folgende Proposition.

Proposition 8.18 *Sei φ ein Constraint. Dann gilt:*

$$\varphi \models\!\mid \overline{\varphi}.$$

Was die Anwendung eines Normierers eines Constraints auf den Constraint für Eigenschaften hat, beschreibt die nächste Proposition.

Proposition 8.19 *Wenn θ ein Normierer von φ ist, dann enthält $\overline{\theta\varphi}$ entweder einen Konflikt, oder ist eine Determinante.*

Beweis Leicht einzusehen ist, daß $\overline{\theta\varphi}$ keine Gleichung $x \doteq y$ enthält: Da θ ein Normierer von φ ist, muß $x = y$ sein, aber triviale Gleichungen werden durch die Operation $\bar{\ }$ bereits entfernt.

Alles weitere sehen wir durch eine Fallunterscheidung ein.

- Sei $\overline{\theta\varphi}$ keine Determinante. Wir zeigen jetzt, daß $\overline{\theta\varphi}$ tatsächlich einen Konflikt enthält.

  Wenn $\bot \in \overline{\theta\varphi}$, dann enthält $\overline{\theta\varphi}$ trivialerweise einen Konflikt. Sei also $\bot \notin \overline{\theta\varphi}$. Da $\overline{\theta\varphi}$ keine Determinante ist, gibt es verschiedene $x \doteq f(\overline{y}), x \doteq f'(\overline{y'}) \in \overline{\theta\varphi}$. Da θ eine Normalisierer von φ ist, und $\overline{\theta\varphi}$ keine Duplikate enthält, muß $f \neq f'$ sein, das heißt $\overline{\theta\varphi}$ enthält einen Konflikt.

- Sei $\overline{\theta\varphi}$ konfliktfrei. Wir zeigen jetzt, daß $\overline{\theta\varphi}$ tatsächlich eine Determinante ist. Seien $x \doteq f(\overline{y}) \in \overline{\theta\varphi}$ und $x' \doteq f'(\overline{y}') \in \overline{\theta\varphi}$ zwei verschiedene Gleichungen. Nun kann aber in keinem Fall $x = x'$ sein, denn sonst enthielte $\overline{\theta\varphi}$ entweder einen Konflikt oder θ wäre kein Normierer von φ. Also ist $\overline{\theta\varphi}$ eine Determinante.

□

Mit Hilfe der obigen Proposition erhalten wir den folgenden Satz.

Satz 8.20 *Sei θ ein Normierer von φ. Dann gelten die beiden folgenden Aussagen:*

1. $\varphi \models\mid_{\mathrm{CET}_0} \theta \wedge \overline{\varphi} \models\mid \theta \wedge \overline{\theta\varphi}$.

2. *φ ist genau dann in* RAT *erfüllbar, wenn $\overline{\theta\varphi}$ keinen Konflikt enthält.*

Beweis

1. Die Regeln (8.1), (8.2) und (8.4) von COL lassen einerseits die Kongruenzen invariant, wie man sich leicht überlegt, andererseits simplifizieren sie φ zu einer Normalform $\theta' \wedge \overline{\delta}$, wobei θ' ein Normierer von $\theta' \wedge \delta$, also auch von φ ist. Damit haben wir dann

$$
\begin{array}{llll}
\varphi & \models\mid_{\mathrm{CET}_0} & \theta' \wedge \overline{\delta} & \\
 & \models\mid & \theta \wedge \overline{\delta} & \text{(Proposition 8.17)} \\
 & \models\mid & \theta \wedge \theta\overline{\delta} & \text{(wegen (8.13))} \\
 & \models\mid & \theta \wedge \overline{\theta\delta} & \text{(Proposition 8.18)}
\end{array}
$$

2. Direkte Folgerung aus der 1. Aussage und den Propositionen 8.19 und 8.12.

□

8.7 Subsumtion in RAT

In diesem Abschnitt wollen wir untersuchen, wie wir Subsumtion und Dissubsumtion von Constraints in dem Constraintsystem RAT entscheiden können.

Bevor wir uns dem eigentlichen Entscheiden der Subsumtionsrelation zuwenden, überlegen wir, in welchem Sinne Subsumtion und Erfüllbarkeit in RAT zusammenhängen.

Proposition 8.21 *Sei δ eine Determinante und σ eine beliebige Formel. Dann gilt:*

$$\delta \models_{\mathrm{RAT}} \sigma \iff \mathrm{RAT} \models \exists\mathcal{D}(\delta)\,(\delta \wedge \sigma)\,.$$

Beweis

„ ⇒ “ Da δ eine Determinante ist, gilt nach dem dritten Axiomenschema von RAT_0

$$\mathrm{RAT}_0 \models \exists \mathcal{D}(\delta)\, \delta, \tag{8.14}$$

und nach Voraussetzung

$$\delta \models_{\mathrm{RAT}} \sigma. \tag{8.15}$$

Also gilt:

$$\begin{array}{lll} \exists \mathcal{D}(\delta)\,(\delta \wedge \sigma) & \models\mid_{\mathrm{RAT}} \;\; \exists \mathcal{D}(\delta)\,\delta & \text{(wegen (8.15))} \\ & \models\mid_{\mathrm{RAT}_0} \;\; \top & \text{(wegen (8.14))} \end{array}$$

„ ⇐ “ Sei $\mathcal{A}$ ein Modell von RAT und α eine Belegung für $\mathcal{A}$ mit $\mathcal{A}, \alpha \models \delta$.
Nach Vorraussetzung gilt

$$\mathcal{A}, \alpha \models \exists \mathcal{D}(\delta)\,(\delta \wedge \sigma)\,.$$

Also existiert eine Belegung β für $\mathcal{A}$, die bis auf die Stellen in $\mathcal{D}(\delta)$ mit α übereinstimmt und für die

$$\mathcal{A}, \beta \models \delta \wedge \sigma$$

gilt. Insbesondere gilt natürlich

$$\mathcal{A}, \beta \models \delta.$$

Da $\mathcal{A}$ ein Modell von RAT ist, also die eindeutige Lösbarkeit von Determinanten erfüllt, muß jetzt $\alpha = \beta$ sein, da $\mathcal{A}, \alpha \models \delta$. Also gilt

$$\mathcal{A}, \alpha \models \sigma,$$

was zu zeigen war.

□

Der nächste Satz gibt eine hinreichende Bedingung dafür, wann eine Determinante einen Normierer subsumiert.

Satz 8.22 *Sei δ eine Determinante und θ Normierer einer Kongruenz von δ, mit $\theta\delta$ konfliktfrei und $\mathcal{V}(\theta) \subseteq \mathcal{D}(\delta)$. Dann gilt:*

$$\delta \models_{\mathrm{RAT}_0} \theta.$$

Beweis Wir argumentieren wie folgt:

$$\begin{array}{llll} \exists \mathcal{D}(\delta)\,(\delta \wedge \theta) & \models\mid & \exists \mathcal{D}(\delta)\,(\theta\delta \wedge \theta) & \text{(wegen (8.13))} \\ & \models\mid & \exists \mathcal{D}(\delta)\,(\theta\delta) & (\theta \text{ idempotent}, \mathcal{D}om(\theta) \subseteq \mathcal{D}(\delta)) \\ & \models\mid_{\mathrm{RAT}_0} & \top & (\theta\delta \text{ Determinante}, \mathcal{D}(\theta\delta) \subseteq \mathcal{D}(\delta)) \end{array}$$

□

In dem obigen Lemma ist wichtig, daß alle Variablen des Normierers durch die Determinante bestimmt werden.

Nach der Behandlung dieses extrem eingeschränkten Falles von Subsumtion, werden wir insbesondere auch den Fall behandeln, bei dem Existenzquantoren auf der linken Seite der Subsumtionsrelation auftreten. An einem Beispiel wollen wir klarmachen, was diese bedeutet.

Beispiel 8.7 Trivial ist die Aussage:

$$\mathsf{X} \doteq \mathsf{f}(\mathsf{Y}) \not\models_{\mathrm{RAT}} \mathsf{X} \doteq \mathsf{f}(\mathsf{Z}),$$

da in der linken Seite nichts über Z ausgesagt wird. Wenn wir aber nur feststellen wollen, ob die Lösung für X in $\mathcal{R}$ mit dem Funktionszeichen f beginnt, so quantifizieren wir Z auf der rechten Seite existentiell.

Aber dann gilt die Aussage:

$$\mathsf{X} \doteq \mathsf{f}(\mathsf{Y}) \models_{\mathrm{RAT}} \exists \mathsf{Z}\, \mathsf{X} \doteq \mathsf{f}(\mathsf{Z}).$$

□

In einem gewissen Sinne sind also auf die der rechten Seite existentiell quantifizierten Variablen für Subsumtion „frei belegbar". Dieses führt uns zu dem folgenden Begriff.

Definition 8.5 (Orientierter Normierer) *Sei $\overline{x}$ eine Folge von Variablen und θ ein Normierer. Dann ist θ ein $\overline{x}$–***orientierter Normierer***, wenn für alle $y \in VAR$ gilt:*

$$\theta(y) \in \mathcal{V}(\overline{x}) \;\Rightarrow\; y \in \mathcal{V}(\overline{x}).$$

Es ist einfach, orientierte Normierer zu erhalten. Sei dazu ρ eine Äquivalenzrelation auf Variablen, zu der es einen Normierer gibt. Das heißt, es gibt nur endlich viele nichttriviale Äquivalenzklassen, die wiederum nur endlich viele Elemente enthalten können. Sei $\{x_1, \ldots, x_n\}$ so eine Äquivalenzklasse. Wenn $\{x_1, \ldots, x_n\} \subseteq \mathcal{V}(\overline{y})$, dann erfüllt die Definition

$$\theta(x_i) := x_1 \qquad i \in \{1, \ldots, n\}$$

alle Anforderungen an einen $\overline{y}$–gerichteten Normierer. Wenn $x_j \notin \mathcal{V}(\overline{y})$ ist, so erfüllt

$$\theta(x_i) := x_j \qquad i \in \{1, \ldots, n\}$$

wiederum alle Anforderungen an einen $\overline{y}$–gerichteten Normierer. Die gleiche Argumentation gilt dann analog auch für die trivialen Äquivalenzklassen von ρ.

Das nächste Lemma wird uns sowohl die nötige Einsicht für die Subsumtion als auch für den Beweis der Unabhängigkeit von RAT liefern. Wir nennen eine Folge von Variablen $\overline{x}$ **linear**, wenn alle Variablen in $\overline{y}$ paarweise verscheiden sind.

Lemma 8.23 (Sabotage) *Sei δ eine Determinante, θ ein $\overline{x}$–orientierter Normierer von $\delta \wedge \varphi$, $\mathcal{V}(\delta) \parallel \mathcal{V}(\overline{x})$ und $\theta(\delta \wedge \varphi)$ konfliktfrei.*

Wenn eine der Bedingungen

(a) $\mathcal{D}(\overline{\theta\varphi}) \subseteq \mathcal{D}(\delta) \cup \mathcal{V}(\overline{x})$,

(b) $\mathcal{D}om(\theta) \subseteq \mathcal{D}(\delta) \cup \mathcal{V}(\overline{x})$,

(c) $\theta(\mathcal{D}(\delta)) \subseteq \mathcal{D}(\delta)$,

verletzt ist, dann ist eine der Aussagen

1. $\exists x, y \in \mathcal{V}(\delta \wedge \varphi) - (\mathcal{D}(\delta) \cup \mathcal{V}(\overline{x}))$: $\quad \neg x \doteq y \wedge \delta \wedge \varphi \models \bot$
2. $\exists x \in \mathcal{V}(\delta \wedge \varphi) - (\mathcal{D}(\delta) \cup \mathcal{V}(\overline{x}))\,, \exists f(\overline{y}), \overline{y}$ *linear*: $\quad \neg\exists\overline{y}\, x \doteq f(\overline{y}) \wedge \delta \wedge \varphi \models_{\mathrm{CET}_0} \bot$

erfüllt.

Beweis Wir nehmen jetzt nach einander an, daß eine der drei Bedingungen (a) bis (c) verletzt ist, und zeigen dann die Gültigkeit einer der beiden Aussagen.

(a) Sei also $x \in \mathcal{D}(\overline{\theta\varphi}) - (\mathcal{D}(\delta) \cup \mathcal{V}(\overline{x}))$. Da $\overline{\theta\varphi}$ Determinante ist, gibt es $x \doteq f(\overline{y}) \in \overline{\theta\varphi}$. Dieses x und f erfüllen also, für jedes geeignete $\overline{z}$,

$$\neg\exists\overline{z}\, x \doteq f(\overline{z}) \wedge \overline{\theta\varphi} \models_{\mathrm{CET}_0} \bot.$$

Damit ist die zweite Aussage erfüllt.

(b) Sei $x \in \mathcal{D}om(\theta) - (\mathcal{D}(\delta) \cup \mathcal{V}(\overline{x}))$. Es gilt $\delta \wedge \varphi \models_{\mathrm{RAT}} x \doteq \theta(x)$. Wir unterscheiden zwei Fälle:

$\theta(x) \notin \mathcal{D}(\delta) \cup \mathcal{V}(\overline{x})$: Dann gilt Aussage 1. mit $y = \theta(x)$.

$\theta(x) \in \mathcal{D}(\delta) \cup \mathcal{V}(\overline{x})$: Da θ $\overline{x}$–orientiert ist, gilt $\theta(x) \in \mathcal{D}(\delta)$. Also gibt es $\theta(x) \doteq f(\overline{z}) \in \delta$, und die zweite Aussage ist erfüllt.

(c) Sei $x \in \theta(\mathcal{D}(\delta)) - \mathcal{D}(\delta)$, also existiert $y \doteq f(\overline{z}) \in \mathcal{D}(\delta)$ mit $\theta(y) = x$.
Da θ $\overline{x}$–orientiert ist, haben wir zudem $x \notin \mathcal{D}(\delta) \cup \mathcal{V}(\overline{x})$. Also gilt die zweite Aussage, da $\delta \wedge \varphi \models_{\mathrm{RAT}} \theta$ und folglich $\delta \wedge \varphi \models_{\mathrm{RAT}} x \doteq f(\overline{z})$.

□

Wir können nun ein Entscheidungsverfahren für Subsumtion und Dissubsumtion angeben. Vorher wollen wir uns jedoch zuerst überlegen, welche Fälle unser Verfahren behandeln muß.

Unser Ausgangspunkt ist, daß wir

$$\exists\overline{x}\, \varphi \models_{\mathrm{RAT}} \exists\overline{y}\, \psi$$

entscheiden wollen. Wenn wir auf beiden Seiten die existentiell quantifizierten Variablen umbenennen, so daß die Variablen in $\overline{x}'$ und $\overline{y}'$ paarweise verschieden sind und nicht frei auf den beiden Seiten vorkommen, so erhalten wir die äquivalente Aussage

$$\exists\overline{x}'\, \varphi' \models_{\mathrm{RAT}} \exists\overline{y}'\, \psi',$$

wobei $\varphi' := \varphi[\overline{x}'/\overline{x}]$ und $\psi' := \psi[\overline{y}'/\overline{y}]$ ist. Diese Aussage können wir jedoch nach (8.11) in Proposition 8.13 zu

$$\varphi' \models_{\mathrm{RAT}} \exists \overline{y}' \varphi'$$

vereinfachen. Jetzt berechnen wir einen Normierer θ von φ'. Wenn $\overline{\theta\varphi'}$ einen Konflikt hat, gilt trivialerweise Subsumtion und Dissubsumtion. Nehmen wir also an, daß $\overline{\theta\varphi'}$ konfliktfrei ist. Dann erhalten wir:

$$\begin{aligned}
\varphi' \models_{\mathrm{RAT}} \exists \overline{y}' \psi' &\iff \theta \wedge \theta\varphi' \models_{\mathrm{RAT}} \exists \overline{y}' \psi' \\
&\iff \theta \wedge \overline{\theta\varphi'} \models_{\mathrm{RAT}} \exists \overline{y}' \psi' \\
&\iff \overline{\theta\varphi'} \models_{\mathrm{RAT}} \exists \overline{y}' \theta\psi'
\end{aligned}$$

Also reicht es, ein Verfahren für Subsumtion anzugeben, daß auf der linken Seite eine Determinante ohne Existenzquantoren hat. Genau das wollen wir im nächsten Satz tun.

Satz 8.24 (Subsumtion und Dissubsumtion in RAT) *Sei δ eine Determinante, $\mathcal{V}(\delta) \parallel \mathcal{V}(\overline{x})$ und θ ein $\overline{x}$-orientierter Normierer von $\delta \wedge \theta$. Dann gilt:*

1. $\delta \models_{\mathrm{RAT}} \neg\exists \overline{x}\, \varphi \iff \theta(\delta \wedge \varphi)$ *enthält Konflikt.*
2. $\delta \models_{\mathrm{RAT}} \exists \overline{x}\, \varphi \iff$ *(a)* $\theta(\delta \wedge \varphi)$ *konfliktfrei,*

   (b) $\mathcal{D}(\overline{\theta\varphi}) \subseteq \mathcal{D}(\delta) \cup \mathcal{V}(\overline{x})$,

   (c) $\mathcal{D}om(\theta) \subseteq \mathcal{D}(\delta) \cup \mathcal{V}(\overline{x})$,

   (d) $\theta(\mathcal{D}(\delta)) \subseteq \mathcal{D}(\delta)$.

Beweis

1. Durch Äquivalenzumformungen erhalten wir

$$\begin{aligned}
\delta \models_{\mathrm{RAT}} \neg\exists \overline{x}\, \varphi &\iff \delta \wedge \exists \overline{x}\, \varphi \models_{\mathrm{RAT}} \bot \\
&\iff \exists \overline{x}\, (\delta \wedge \varphi) \models_{\mathrm{RAT}} \bot && (\mathcal{V}(\delta) \parallel \mathcal{V}(\overline{x})) \\
&\iff \exists \overline{x}\, (\theta \wedge \theta\delta \wedge \theta\varphi) \models_{\mathrm{RAT}} \bot && (\theta \text{ Normierer})
\end{aligned}$$

   Die letzte Aussage ist aber nach Satz 8.20 äquivalent dazu, daß $\overline{\theta(\delta \wedge \varphi)}$ einen Konflikt enthält, also muß auch $\theta(\delta \wedge \varphi)$ einen Konflikt enthalten.

2. Nach Proposition 8.21 ist die Aussage $\delta \models_{\mathrm{RAT}} \exists \overline{x}\, \varphi$ äquivalent zu der Aussage

$$\mathrm{RAT} \models \exists \mathcal{D}(\delta)\, (\delta \wedge \exists \overline{x}\, \varphi)\,.$$

   „$\Leftarrow$“ Nachrechnen ergibt:

$$\begin{aligned}
&\exists \mathcal{D}(\delta)\, (\delta \wedge \exists \overline{x}\, \varphi) \\
&\models\!\!\mid \quad \exists \mathcal{D}(\delta) \exists \overline{x}\, (\delta \wedge \varphi) && (\mathcal{V}(\delta) \parallel \mathcal{V}(\overline{x})) \\
&\models\!\!\mid_{\mathrm{CET}_0} \quad \exists \mathcal{D}(\delta) \exists \overline{x} \left(\theta \wedge \theta\delta \wedge \overline{\theta\varphi}\right) && (\theta \text{ Normierer}) \\
&\models\!\!\mid \quad \exists \mathcal{D}(\delta) \exists \overline{x}\, \underbrace{\left(\theta\delta \wedge \overline{\theta\varphi}\right)}_{\text{Determinante}} && ((c) \text{ mit Proposition 8.13}) \\
&\models\!\!\mid_{\mathrm{RAT}_0} \quad \top && ((a), (b) \text{ und } (d))
\end{aligned}$$

   Also folgt mit Proposition 8.21 die Behauptung.

„ $\Rightarrow$ “ Die Konfliktfreiheit (Bedingung (a)) ist einfach einzusehen. Mit Proposition 8.21 wissen wir, daß

$$\mathrm{RAT} \models \exists \mathcal{D}(\delta) \exists \overline{x} \left(\theta \wedge \theta\delta \wedge \overline{\theta\varphi} \right)$$

gilt. Das heißt nach dem Sabotage Lemma 8.23, daß es kein $x \in \mathcal{D}(\overline{\theta\varphi})$ gibt, daß nicht existentiell quantifiziert ist, also gilt Bedingung (b). Aber auch die Bedingungen (c) und (d) sind erfüllt: beidesmal sagt das wiederum das Sabotage Lemma 8.23 aus.

□

8.8 Unabhängigkeit von RAT

Dank der Vorarbeit durch das Sabotage Lemma 8.23 können wir einfach die Unabhängigkeit von RAT gewinnen. Man beachte, daß diese Formulierung stärker ist, als in der Definition 4.2 verlangt, da hier auch Existenzquantoren in der Disjunktion erlaubt sind.

Satz 8.25 (Unabhängigkeit von RAT) *Seien $\varphi, \psi_1, \ldots, \psi_n$ Constraints. Dann gilt:*

$$\exists \overline{x}\, \varphi \models_{\mathrm{RAT}} \bigvee_{i=1}^{n} \exists \overline{y}_i\, \psi_i \;\Rightarrow\; \exists i \in \{1, \ldots, n\} : \quad \exists \overline{x} \models_{\mathrm{RAT}} \exists \overline{y}_i\, \psi_i. \tag{8.16}$$

Beweis Analog zu der Argumentation bei der Subsumtion in RAT, daß es genügt auf der linken Seite der Relation $\models_{\mathrm{RAT}}$ nur eine Determinante zu betrachten, können wir uns darauf beschränken, anstatt (8.16) die folgende Aussage zu zeigen:

$$\delta \models_{\mathrm{RAT}} \bigvee_{i=1}^{n} \exists \overline{y}_i\, \psi_i \;\Rightarrow\; \exists i \in \{1, \ldots, n\} : \quad \delta \models_{\mathrm{RAT}} \exists \overline{y}_i\, \psi_i,$$

wobei δ eine Determinante und $\psi_1, \ldots, \psi_n$ Constraints sind.

O.B.d.A. seien alle $\delta \wedge \exists \overline{y}_i\, \varphi$ erfüllbar in RAT, und die $\overline{y}_i$ so gewählt, daß für alle $i, j \in \{1, \ldots, n\}$ mit $i \neq j$ gilt:

$$\mathcal{V}(\overline{y}_i) \parallel \mathcal{V}(\overline{y}_j), \qquad \mathcal{V}(\delta) \parallel \mathcal{V}(\overline{y}_i), \qquad (\mathcal{V}(\psi_i) - \mathcal{V}(\overline{y}_i)) \parallel \mathcal{V}(\overline{y}_j).$$

Sei θ_i ein $\overline{y}_i$–orientierter Normierer von $\delta \wedge \psi_i$. Dann sind aber nach unserer Eingangs gemachten Annahme alle $\theta_i(\delta \wedge \psi_i)$ konfliktfrei.

Wir führen den Beweis durch Widerspruch und machen die Annahme

$$\forall i \in \{1, \ldots, n\} : \quad \delta \not\models_{\mathrm{RAT}} \exists \overline{y}_i\, \psi_i. \tag{8.17}$$

Damit können wir für alle i das Sabotage Lemma 8.23 anwenden. Also existiert eine Substitution ω mit

$$\mathcal{D}om(\omega) \parallel \left(\mathcal{D}(\delta) \cup \bigcup_{i=1}^{n} \mathcal{V}(\overline{y}_i) \right)$$

und

$$\bigvee_{i=1}^{n} (\omega\delta \wedge \omega\psi_i) \models_{\mathrm{CET}_0} \bot.$$

Da $\delta \models_{\mathrm{RAT}} \bigvee_{i=1}^{n} \exists\overline{y}_i\, \psi_i$ gilt nach Proposition 8.21

$$\begin{aligned} \top \quad & \models\!\mid_{\mathrm{RAT}} \quad \exists\mathcal{D}(\delta) \left(\delta \wedge \bigvee_{i=1}^{n} \exists\overline{y}_i\, \psi_i\right) \\ & \models\!\mid \qquad \exists\mathcal{D}(\delta) \bigvee_{i=1}^{n} \exists\overline{y}_i\, (\delta \wedge \psi_i) \end{aligned}$$

also

$$\begin{aligned} \top \quad & \models\!\mid_{\mathrm{RAT}} \quad \omega\left(\exists\mathcal{D}(\delta) \bigvee_{i=1}^{n} \exists\overline{y}_i\, (\delta \wedge \psi_i)\right) \\ & \models\!\mid \qquad \exists\mathcal{D}(\delta) \bigvee_{i=1}^{n} \exists\overline{y}_i\, (\omega\delta \wedge \omega\psi_i) \\ & \models\!\mid \qquad \exists\mathcal{D}(\delta)\exists\overline{y}_1 \cdots \exists\overline{y}_n \bigvee_{i=1}^{n} (\underbrace{\omega\delta \wedge \omega\psi_i}_{\models_{\mathrm{CET}_0} \bot}) \\ & \models\!\mid_{\mathrm{CET}_0} \quad \bot \end{aligned}$$

Aber das ist ein Widerspruch, somit ist die Annahme 8.17 falsch. $\square$

Teil III

Abstrakte Maschinen

Kapitel 9

Die RAT–Maschine

In Kapitel 8 haben wir das Constraintsystem RAT kennengelernt und die theoretischen Aspekte, die mit Erfüllbarkeit und Subsumtion zu tun haben, kennengelernt. In diesem Kapitel wollen wir jedoch eher die praktische Seite beleuchten, indem wir eine abstrakte Maschine entwerfen, die dazu in der Lage ist, Constraints von RAT zu verarbeiten.

9.1 Ausgangspunkt

Die Idee einer abstrakten Maschine ist wie folgt: Wir gehen von einer Zielsprache aus, die im allgemeinen nicht dazu geeignet ist, direkt auf einem Computer ausgeführt zu werden. Diese Sprache übersetzen wir in eine Sequenz von abstrakten Maschineninstruktionen. Schließlich werden diese Instruktionen dann von der abstrakten Maschine ausgeführt, wobei die abstrakte Maschine selber ein Programm auf einer konkreten Maschine ist, das die auszuführenden Instruktionen abarbeitet.

Konkret für unseren Zweck bedeutet das:

1. Der Benutzer gibt einen RAT–Constraint ein.
2. Der Übersetzer übersetzt den eingegebenen Constraint in eine Sequenz von RAT–**Maschineninstruktionen**.
3. Die RAT–**Maschine** arbeitet diese Sequenz von Instruktionen ab. Danach werden Meldungen ausgegeben, wie zum Beispiel „Constraint war erfüllbar“ oder „Constraint ist dissubsumiert“.
4. Der Benutzer hat erneut die Möglichkeit, bei 1. weiterzumachen.

Diese Art, der Maschine inkrementell neue Eingaben zu präsentieren, die diese dann abarbeitet, wird auch „**On–Line**“ Betrieb genannt, im Gegensatz zum „**Off–Line**“ Betrieb, wo es dem Benutzer nur erlaubt ist, einmal einen Constraint anzugeben, den die Maschine dann abarbeitet, ohne die Möglichkeit weitere Constraint hinzuzufügen.

Wie wir uns leicht überlegen, wird es essentiell für die Einfachheit der Maschine sein, daß die bereits konsumierten Constraints in einer kanonischen Darstellung repräsentiert werden, so daß weitere Berechnungen einfach und effizient ablaufen können.

Also werden wir in der Maschine nur kanonische Constraints repräsentieren. Dann erbringt die Maschine also die Leistung

$$\mathit{Kanonischer\ Constraint} \wedge \left\{ \begin{array}{c} x \doteq y \\ x \doteq f(\overline{y}) \end{array} \right\} \leadsto \left\{ \begin{array}{c} \mathit{Kanonischer\ Constraint} \\ \bot \end{array} \right\}.$$

Der erste Punkt des Entwurfs wird also sein, zu überlegen, wie sich kanonische Constraints in der Maschine darstellen lassen.

9.2 Darstellung von Constraints

Das erste, was es anzugeben gilt, ist, wie die Datentypen für die Repräsentation von Bäumen in der abstrakten Maschine aussehen. Eine Datentypdeklaration hierfür ist in Abbildung 9.1 zu sehen. Hier und auch im weiteren werden wir uns einer PASCAL–ähnlichen Darstellungsform bedienen, wo wir uns jedoch erlauben, manche Sachverhalte verkürzt darzustellen. Zusätzlich zur Typdeklaration sind die beiden Varianten des Datentyps skizziert. Wir werden hier und im weiteren *ref*–Zeiger immer grau wiedergeben. In weiteren Beispielen wird das *tag*–Feld einfach unterschlagen, da es immer aus dem Kontext ersichtlich sein wird, und Symbol und Arität werden zusammen dargestellt.

Elemente des Datentyps *cell* werden durch das Feld *tag* in Variablen (*tag* = `var`) und Strukturen (*tag* = `struct`) unterschieden. Wir werden diese als **Zellen** und **Variablenzellen**, respektive **Strukturzellen**, bezeichnen

Wenn es sich bei einem Element `Cell` um eine Variable handelt, so kann die Variable möglicherweise ungebunden sein, in diesem Fall gilt, daß `Cell.`*ref* gleich **`nil`** ist. Andernfalls ist die Variable an die Darstellung eines anderen Terms gebunden, daß heißt `Cell.`*ref* zeigt auf die Zelle, an die die Variable gebunden ist.

Ist `Cell` eine Struktur, also die Codierung des Kopfes $f(\cdots)$ eines Terms $f(x_1, \ldots, x_n)$, so beschreibt das Feld *sym* das Symbol f der Struktur (Die Modellierung durch den Datentyp **`nat`** macht die Annahme, daß diese Symbole bereits vom Übersetzer in eine Symboltabelle eingetragen wurden, und auf diese Symbole eindeutig durch natürliche Zahlen zugegriffen werden kann). Die Arität n wird durch das Feld *arity* gegeben, das Feld *child*[i] zeigt auf den i–ten Unterbaum. Wir nehmen hier an, daß das Feld *child* gerade genausoviel Elemente hat, wie durch *arity* vorgegeben. Den Zweck des Feldes *ref* bei einer Struktur werden wir später noch diskutieren.

In Abbildung 9.1 sind zusätzlich noch zwei Prozeduren angegeben, die neue Elemente vom Datentyp *cell* im Speicher der abstrakten Maschine erzeugen und initialisieren.

9.3 Grundlegende Funktionalität

Im folgenden werden wir untersuchen, was die Maschine leisten muß, um die Constraints

$$\mathsf{X} \doteq \mathsf{f}(\mathsf{X},\mathsf{Z}) \tag{9.1}$$
$$\mathsf{Y} \doteq \mathsf{f}(\mathsf{Z},\mathsf{Y}) \tag{9.2}$$
$$\mathsf{Y} \doteq \mathsf{X} \tag{9.3}$$

```
type cell = record
      case tag of
         var    : ref   : ↑cell ;
         struct: sym   : nat;
                 arity : nat;
                 child : array [nat] of ↑cell ;
                 ref   : ↑cell
      esac
      end;

procedure new_var(var X: ↑cell )
   new(X);
   X↑ := ⟨tag : var,ref : nil⟩
end new_var;

procedure new_struct(F, N: nat; var X: ↑cell )
   new(X);
   X↑ := ⟨tag : var, sym : F, arity : N, ref : nil⟩
end new_struct;
```

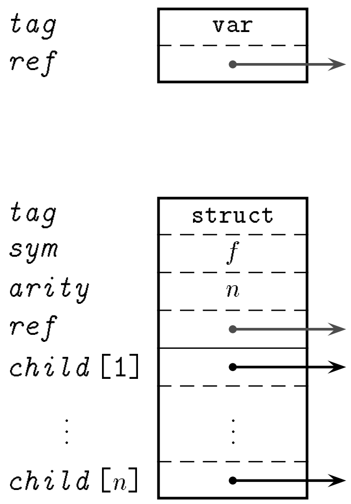


Abbildung 9.1: Datentypen für RAT–Constraints

einzulesen, abzuarbeiten, und eine kanonische Darstellung in der Maschine zu erzeugen.

Der Übersetzer wird für die Constraints (9.1) bis (9.3) Instruktionen erzeugen, die nach Abarbeitung die kanonische Repräsentation der Constraints in der Maschine realisiert hat. Dazu wird die Maschine Register haben, in dem Referenzen auf die in der Maschine gespeicherten Daten enthalten sind. Der Übersetzer wird die Register sukzessive für die zu erzeugenden Variablen und Strukturen vergeben. Bezeichnen werden wir diese Register mit `X`, die Deklaration lautet somit:

```
X: array [nat] of ↑cell
```

Zunächst muß also der Constraint $\mathsf{X} \doteq \mathsf{f}(\mathsf{X}, \mathsf{Z})$ in Instruktionen übersetzt werden, und diese ausgeführt werden. Dabei werden wir zuerst die Argumente des Terms $\mathsf{f}(\mathsf{X}, \mathsf{Z})$ durch geeignete Instruktionen erzeugen, und dann den eigentlichen Constraint realisieren.

Das Erzeugen einer neuen Variablen übernimmt die Instruktion `put_var X`i, die mittels der Operation `new_var(X[`i`])` eine neue Variable erzeugt und eine Referenz auf diese im Register `X`i ablegt. Wir schreiben für das Register i im weiteren suggestiver `X`i.

Aber die Maschine muß nicht nur neue Variablen erzeugen, sondern auch neue Strukturen. Dieses übernimmt die Instruktion `put_struct X`i, f, n, die im Register `X`i einen Strukturknoten mit dem Symbol f und der Arität n einrichtet. Um jedoch eine komplette Struktur zu erzeugen, müssen natürlich auch noch die n Argumente der Struktur erzeugt werden. Dieses geschieht dadurch, daß die `put_struct`–Instruktion die globalen Variablen `Struct: ↑cell` und `Arg: nat` initialisiert, so daß mittels `Struct↑.child[Arg]` auf das erste Argument der Struktur zugegriffen werden kann. Das eigentliche Erzeugen

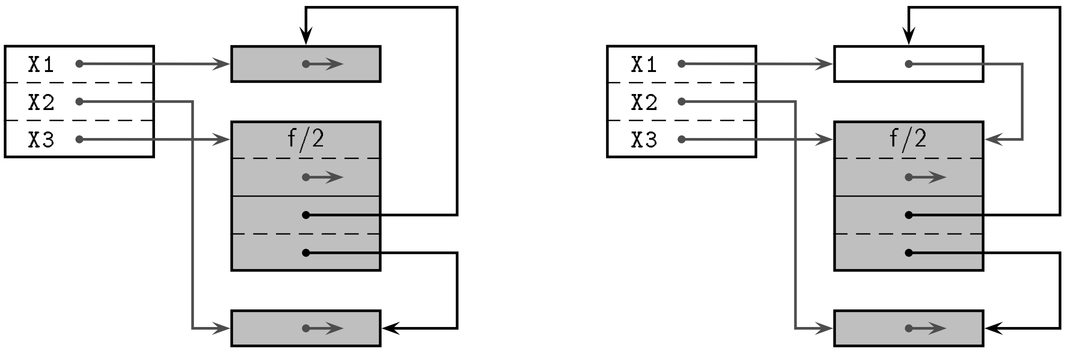


Abbildung 9.2: Zustand nach Erzeugen und Binden

der Argumente der Struktur geschieht nun wie folgt: Für jedes Argument wird durch eine Instruktion `put_arg X`i das Feld `Struct`$\uparrow$`.`*child*`[Arg]` auf `X[`i`]` gesetzt, und danach der Zähler `Arg` um eins inkrementiert, um auf das nächste Argument Zugriff zu erhalten.

Wir nehmen an, daß der Übersetzer die Referenzen auf die Variablen X und Z in den Registern `X1`, respektive `X2`, belegt, und für das Erzeugen der Struktur das Register `X3`. Also haben wir für das Erzeugen des Terms $\mathsf{f}(\mathsf{X}, \mathsf{Z})$ die folgende Instruktionssequenz:

```
put_var      X1          %  Erzeuge X
put_var      X2          %  Erzeuge Z
put_struct   X3, f, 2    %  Erzeuge f(
put_arg      X1          %             X,
put_arg      X2          %             Z)
```

Der sich ergebende Maschinenzustand nach Ausführung dieser Instruktionssequenz ist in Abbildung 9.2 im linken Teilbild skizziert.

Um den Constraint $\mathsf{X} \doteq \mathsf{f}(\mathsf{X}, \mathsf{Z})$ vollständig zu realisieren, müssen wir die Variablenzelle, auf die das Register `X1` zeigt, mit der gerade erzeugten Strukturzelle unifizieren. Dieses wird die Instruktion

```
unify  X1, X3      %  Unifiziere X mit f(X, Z)
```

durchführen. Wie dieses vor sich geht, ist einfach einzusehen: Die Variable, auf die das Register `X1` zeigt, wird an die Struktur, auf die das Register `X3` zeigt, gebunden. Die „Bindung" wird dadurch hergestellt, daß das *ref*–Feld der Variablenzelle auf die Strukturzelle gesetzt wird. Der sich ergebende Maschinenzustand nach dem Durchführen dieser Operation ist im rechten Teilbild von Abbildung 9.2 zu sehen. Die genaue Definition der `unify`–Operation wollen wir auf später verschieben, wir wissen aber bereits, was geschieht, wenn eine Variable mit einer Struktur unifiziert wird.

In der obigen Argumentation sind wir implizit davon ausgegangen, daß wir an Hand einer Referenz eine Zelle erhalten können. Aber auch die Maschine muß in der Lage sein, einer Referenz zu folgen und die erste Zelle zu erhalten, deren *ref*–Feld gleich **`nil`** ist. Im weiteren werden wir Zellen, deren *ref*–Feld gleich **`nil`** ist, als **echte Zellen**

```
procedure deref(var X: ↑cell)
   while X↑.ref ≠ nil do X := X↑.ref od
end deref;
```

Abbildung 9.3: Die Dereferenzierungsoperation

bezeichnen, und echte Zellen sind in Abbildungen immer hellgrau unterlegt. In der Skizze des Maschinenzustandes nach Ausführen der Instruktion `unify X1, X3` (also die rechte Seite von Abbildung 9.2) ist zu erkennen, daß die echte Zelle, auf die das Register `X1` zeigt, erst nach zweimaligem Dereferenzieren erhalten werden kann.

Wir brauchen also eine Operation, die Referenzketten abläuft und zu einer gegebenen Referenz die erste echte Zelle liefert. Diese Operation ist in Abbildung 9.3 angegeben. Insbesondere folgt diese Operation auch Referenzketten, die durch Strukturzellen verlaufen, der Sinn hiervon wird später klar werden.

Die Übersetzung des zweiten Constraints $\mathsf{Y} \doteq \mathsf{f(Z,Y)}$ erfolgt im wesentlichen wie oben schon ausgeführt, nur müssen wir diesmal keine Variablenzelle für Z anlegen, da diese bereits existiert, und Register `X2` zeigt auf diese. Als Instruktionssequenz erhalten wir also:

```
put_var      X4          % Erzeuge Y
put_struct   X5, f, 2    % Erzeuge f(
put_arg      X2          %           Z,
put_arg      X4          %           Y)
unify        X4, X5      % Unifiziere Y mit f(Z,Y)
```

Der verbleibende Constraint $\mathsf{Y} \doteq \mathsf{X}$ wird uns den Rest der `unify`–Operation einsichtig machen. Wir müssen in der Maschine jetzt die beiden Register `X1` und `X4` miteinander unifizieren. Dazu wird die Maschine die folgende Instruktion ausführen:

```
unify  X1, X4     % Unifiziere X mit f(X,Z)
```

Der Ausgangszustand der Maschine ist jetzt in Abbildung 9.4 oben dargestellt, wobei die `X`–Register aus Gründen der Übersichtlichkeit bis auf die beiden uns interessierenden unterdrückt wurden.

Das erste, was die Maschine zu tun hat, ist durch Dereferenzieren die beiden echten Zellen, also die beiden Strukturzellen, zu erhalten. Nachdem diese sichtbar sind, muß überprüft werden, ob die Symbole und die Aritäten der beiden Strukturzellen übereinstimmen. In unserem Beispiel ist das der Fall, aber sonst müßte die Maschine anhalten und ausgeben, daß keine Erfüllbarkeit vorliegt.

Die grobe Richtung, in die sich die Aktivitäten der Maschine bewegen werden, liegt klar auf der Hand: Die Argumente der beiden Strukturzellen müssen miteinander unifiziert werden. Aber da wir ja zyklische Strukturen in der Maschine haben (siehe Abbildung 9.2), muß die Maschine bei der rekursiven Unifikation der Argumente sicherstellen, daß sie in keine Endlosschleife läuft.

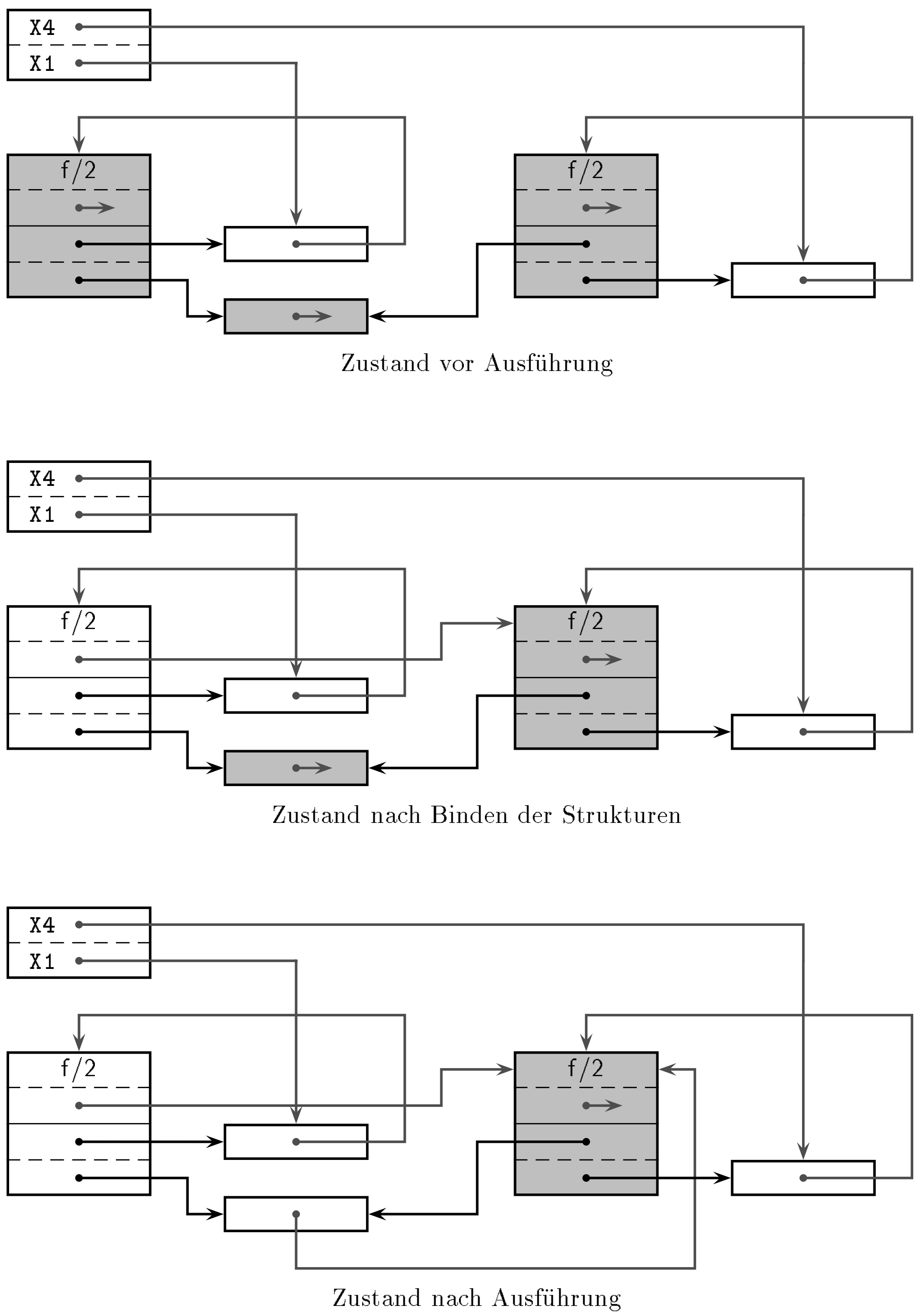


Abbildung 9.4: Zustände bei Ausführung von `unify(X[1], X[4])`

```
procedure unify(X, Y: ↑cell)
   deref(X); deref(Y);
   if X ≠ Y
   then if X↑.tag = var
        then X↑.ref ← Y
        else if Y↑.tag = var
             then Y↑.ref ← X
             else if X↑.sym = Y↑.sym ∧ X↑.arity = Y↑.arity
                  then X↑.ref := Y;
                       for I ← 1 to X↑.arity do
                          unify(X↑.child[I], Y↑.child[I])
                       od
                  else goto failure
                  fi
             fi
        fi
   fi
end unify;
```

Abbildung 9.5: Die Unifikationsprozedur

Die Technik, der wir uns bedienen werden, ist einfach: Bevor die Maschine rekursiv mit der Unifikation der Argumente beginnt, setzt sie das *ref*–Feld eines der beiden zu unifizierenden Strukturzellen auf die andere Strukturzelle. Wenn die Maschine zusätzlich berücksichtigt, daß es nicht nötig ist, bereits identische Zellen weiter rekursiv miteinander zu unifizieren, hat die obige Technik die folgenden Effekte: Bei jedem rekursiven Aufruf nimmt die Anzahl der echten Zellen echt ab. Weiterhin erfüllt die Maschine die Invariante, daß die Verkettung durch die *ref*–Felder azyklisch ist. Also muß die Maschine terminieren.

An unserem Beispiel wollen wir jetzt diese Technik studieren. Nach dem Binden der ersten Strukturzelle an die zweite, haben wir einen Zustand, wie in der Mitte der Abbildung 9.4 skizziert. Dann setzt die Maschine die Unifikation der Argumente rekursiv fort.

Um die beiden ersten Argumente zu unifizieren, werden diese zuerst dereferenziert. Das Dereferenzieren des ersten Argumentes ergibt die Struktur, auf die Register `X4` zeigt, in der Abbildung 9.4 also jeweils die rechte. Die einzige noch ungebundene Variable liefert das Dereferenzieren des anderen ersten Argumentes. Folglich wird die ungebundene Variable jetzt an die durch Dereferenzieren gewonnene Struktur gebunden.

Danach terminiert der rekursive Aufruf der Unifikation für die ersten Argumente, und die Unifikation wird bei den zweiten Argumenten fortgesetzt. Hier gibt es aber nichts mehr zu tun, da das Dereferenzieren beider Argumente die Struktur gibt, auf die das Register `X4` zeigt, und nach unserer Vorüberlegung somit nichts mehr zu tun ist. Also gibt der untere Teil der Abbildung 9.4 den Maschinenzustand nach Ausführung aller Instruktionen an, die zur Realisierung der Constraints (9.1) bis (9.2) nötig waren.

```
var X      : array [nat] of ↑cell ;
    Code   : array [nat] of Instruction;
    PC     : nat := 1;
    Struct: ↑cell ;
    Arg    : nat;

emulate : case Code[PC] of
            ⟨put_struct, I, F, N⟩: new_struct(X[I], F, N);
                                   Struct := X[I];
                                   Arg := 1;
            ⟨put_var, I⟩         : new_var(X[I]);
            ⟨put_arg, I⟩         : Struct↑.child [Arg] := X[I];
                                   Arg := Arg + 1;
            ⟨unify, I, J⟩        : unify(X[I], X[J]);
            ⟨end⟩                : return ''execution halted''
          esac;
          PC := PC + 1;
          goto emulate ;

failure : return ''execution failed'';
```

Abbildung 9.6: Register und Emulatorschleife der Maschine

Jetzt ist für uns auch einfach, die komplette Unifikationsprozedur anzugeben, diese ist in Abbildung 9.5 dargestellt. In unserem Beispiel hätte also der Aufruf `unify(X[1], X[4])` die oben geschilderten Schritte zur Folge gehabt. Der Sprung nach *failure* ist die oben erwähnte Aktion, die bei Nichtunifizierbarkeit ausgeführt werden soll.

9.4 Die Emulatorschleife

Bis jetzt haben wir nur die Unifikationsprozedur komplett vorgestellt. Wir müssen also einerseits noch die Details der anderen Operationen nachtragen, andererseits müssen wir auch näher auf den „Maschinen"-Aspekt eingehen.

Wir versehen die Maschine jetzt noch um einen **Programmzähler** `PC`, der jeweils auf die gerade abzuarbeitende Maschineninstruktion zeigen wird. Das Programm sei in einem Feld `Code` gegeben. Am Anfang wird die Maschine mit dem Programmzählerwert `PC := 1` gestartet, und die einzelnen Instruktionen werden solange abgearbeitet, bis entweder die Instruktion `end` ausgeführt wird, oder aber bei der Abarbeitung einer `unify`–Instruktion Nichtunifizierbarkeit festgestellt wird, also *failure* angesprungen wird. Zusammengefaßt hat also die Maschine die Struktur, wie sie in Abbildung 9.6 skizziert ist.

9.5 Verwendung existierender Strukturen

In dem Abschnitt 9.3 haben wir für die dort gegebenen Constraints einfachen, aber ineffizienten Code erzeugt. In diesem Abschnitt wollen wir zwei wichtige Optimierungen erläutern, wobei eine auch Eingang in die Maschine halten wird.

Die erste Optimierung fällt sofort auf, wenn wir noch einmal die Instruktionssequenzen betrachten, die wir für die Constraints (9.1) und (9.2) erzeugt haben: Wir haben jeweils für die Variablen X und Y neue Variablen erzeugt, und diese dann sofort an die erzeugten Strukturen gebunden. Diesen Zwischenschritt können wir uns aber sparen. Anstatt eine neue Variable zu erzeugen, werden wir bei den `put_arg`–Instruktionen jeweils das Register verwenden, in dem die Struktur erzeugt wird. Dieses ist möglich, da die Strukturen bereits erzeugt sind, wenn die `put_arg`–Instruktionen von der Maschine abgearbeitet werden.

Damit kommen wir dann für die drei Constraints (9.1) bis (9.3) zu der folgenden Instruktionssequenz:

```
put_var      X2          %  Erzeuge Z
put_struct   X1, f, 2    %  Erzeuge X ≐ f(
put_arg      X1          %              X,
put_arg      X2          %              Z)
put_struct   X4, f, 2    %  Erzeuge Y ≐ f(
put_arg      X2          %              Z,
put_arg      X4          %              Y)
unify        X1, X4      %  Unifiziere X mit Y
```

Insbesondere werden in der obigen Instruktionssequenz die Register `X3` und `X5` nicht mehr verwendet.

Jetzt werden wir uns einer anderen, wesentlichen Optimierung der Maschine zuwenden. Wenn wir daran denken, die bisherigen Operationen unserer Maschine in eine abstrakte Maschine für eine logische Programmiersprache zu integrieren, wird bei der Benutzung der Instruktionen wichtig sein, daß bereits existierende Strukturen in der Maschine soweit wie möglich wieder benutzt werden. Was das genau heißt werden wir jetzt an einem kleinen Beispiel betrachten.

Die rekursive Klausel eines Prädikates in der Programmiersprache Prolog, das zwei Listen konkateniert, lautet üblicherweise:

$$\mathsf{append}(\mathsf{cons}(\mathsf{X},\mathsf{Xr}),\mathsf{Ys},\mathsf{cons}(\mathsf{X},\mathsf{Zr})) \leftarrow \mathsf{append}(\mathsf{Xr},\mathsf{Ys},\mathsf{Zr}).$$

Oder in einer Schreibweise, die die Constraints, an denen wir interessiert sind, hervorhebt:

$$\begin{array}{rcl}\mathsf{append}(\mathsf{Xs},\mathsf{Ys},\mathsf{Zs}) & \leftarrow & \mathsf{Xs} \doteq \mathsf{cons}(\mathsf{X},\mathsf{Xr}) \\ & \wedge & \mathsf{Zs} \doteq \mathsf{cons}(\mathsf{Z},\mathsf{Zr}) \\ & \wedge & \mathsf{X} \doteq \mathsf{Z} \\ & \wedge & \mathsf{append}(\mathsf{Xr},\mathsf{Ys},\mathsf{Zr}).\end{array}$$

Üblicherweise wird nun bei einem Aufruf dieser Klausel wenigstens eine der Listen Xs oder Zs bereits im Speicher einer abstrakten Maschine realisiert sein. Wenn wir mit der uns bisher zur Verfügung stehenden Technik Instruktionen erzeugen wollten, so würden wir unter der Annahme, daß die Registerbelegung für Variablen wie folgt ist:

Variable	Register	Variable	Register	Variable	Register
Xs	X1	X	X4	Xr	X5
Zs	X3	Z	X7	Zr	X8

die folgende Instruktionssequenz erzeugen:

```
put_var     X4            %  Erzeuge X
put_var     X5            %  Erzeuge Xr
put_struct  X6, cons, 2   %  Erzeuge cons(
put_arg     X4            %              X,
put_arg     X5            %              Xr)
unify       X1, X6        %  Unifiziere Xs mit cons(X, Xr)
put_var     X7            %  Erzeuge Z
put_var     X8            %  Erzeuge Zr
put_struct  X9, cons, 2   %  Erzeuge cons(
put_arg     X7            %              Z,
put_arg     X8            %              Zr)
unify       X3, X9        %  Unifiziere Zs mit cons(Z, Zr)
unify       X4, X7        %  Unifiziere X mit Z
```

Bei Abarbeitung dieser Instruktionssequenz würden jeweils zwei komplett neue cons–Zellen mit jeweils zwei neuen Variablenzellen als Argumenten erzeugt. Unter der Annahme, daß zum Beispiel die Variable Xs bereits an eine Liste gebunden ist, ist das komplette Speicherverschwendung, da die erste `unify`–Instruktion dafür sorgt, daß nur einer der beiden jetzt existierenden cons–Strukturen im Maschinenzustand sichtbar bleibt: Entweder die neu erzeugte, oder die schon vorher existierende.

Die Idee für die entsprechende Optimierung ist einfach. Wenn Xi bereits an eine Zelle gebunden ist, wir die Maschine, anstatt erst eine Struktur zu erzeugen, und diese dann zu unifizieren, um die Instruktion `get_struct` Xi, f, n erweitert. Diese Instruktion überprüft, ob in der dereferenzierten Zelle von Xi die Struktur mit dem Symbol f und der Arität n schon erzeugt ist. Ist dies der Fall, wird keine neue Struktur erzeugt, sondern der `Struct`–Zeiger wird auf diese bereits existierende Struktur gesetzt. Andernfalls wird eine neue Struktur den geforderten Angaben nach erzeugt.

Je nachdem welcher Fall vorlag, hat das Ablegen der Argumente unterschiedlich zu erfolgen. Somit sehen wir in der Maschine einerseits eine neue Variable `Put` vom Typ **`bool`** vor, die auf **`true`** gesetzt wird, wenn eine neue Struktur erzeugt wurde. Andernfalls wird `Put` auf **`false`** gesetzt. Man sagt auch, daß die Maschine sich im **Lesemodus** oder im **Schreibmodus** befindet, je nachdem ob `Put` den Wert **`false`** oder **`true`** hat. Das Ablegen der Argumente wird jetzt durch die neue Instruktion `get_arg` Xi vorgenommen, die für den Fall, daß eine Struktur erzeugt wurde, sich gerade wie `put_arg` Xi verhält. Sonst wird diese Instruktion das gerade betrachtete Argument mit Xi unifizieren.

Die beiden `get`–Instruktionen sind in ihrer Integration in die Emulatorschleife in Abbildung 9.7 dargestellt.

```
var Put: bool;

emulate: case Code[PC] of
           ⟨get_struct, I, F, N⟩: deref(X[I]);
                                  if X↑.tag= var
                                  then new_struct(X[I], F, N);
                                       Put := true
                                  else if X[I]↑.sym = F ∧ X[I]↑.arity = N
                                       then Put := false
                                       else goto failure
                                       fi
                                  fi;
                                  Struct := X[I];
                                  Arg := 1;
           ⟨get_arg, I⟩          : if Put
                                  then Struct↑.child[Arg] := X[I]
                                  else unify(X[I], Struct↑.child[Arg])
                                  fi
                                  Arg := Arg + 1;
             ⋮;
         esac;
  ⋮
```

Abbildung 9.7: Erweiterung der RAT–Maschine um get–Instruktionen

Unter Verwendung dieser neuen Instruktion erhalten wir die folgende Instruktionssequenz zu unserem Beispiel:

```
put_var     X4          % Erzeuge X
put_var     X5          % Erzeuge Xr
get_struct  X1, cons, 2 % Beschaffe cons(
get_arg     X4          %              X,
get_arg     X5          %              Xr)
put_var     X7          % Erzeuge Z
put_var     X8          % Erzeuge Zr
get_struct  X3, cons, 2 % Beschaffe cons(
get_arg     X7          %              Z,
get_arg     X8          %              Zr)
unify       X4, X7      % Unifiziere X mit Z
```

Kapitel 10

Die K(RAT)–Maschine

Im letzten Kapitel haben wir eine abstrakte Maschine betrachtet, die dazu in der Lage war, RAT–Constraints entgegenzunehmen und zu einem kanonischen Constraint zu simplifizieren.

In diesem Kapitel werden wir einerseits diese Maschine so erweitern, daß sie Subsumtion und Dissubsumtion von Constraints feststellen kann. Andererseits werden wir das in Kapitel 5 eingeführte Konditional implementieren, somit eine abstrakte Maschine für den Kalkül K(RAT) entwerfen.

10.1 Subsumtion in RAT ohne Existenzquantoren

In diesem Abschnitt werden wir die RAT–Maschine so erweitern, daß sie feststellen kann, ob die bisher eingelesenen Constraints einen anderen Constraint subsumieren oder dissubsumieren.

Aus Benutzersicht wird die Erweiterung so aussehen, daß es erlaubt ist, anstatt eines normalen Constraints φ auch die Eingabe

$$\mathsf{ask}\,[\varphi]$$

zu machen. Der Übersetzer wird diese Eingabe dann wiederum in RAT–Maschineninstruktionen übersetzen.

Die Maschine wird diese Instruktionen abarbeiten, und entsprechende Mitteilungen machen, sie bietet also die folgende neue Funktionalität an:

$$\textit{Kanonischer Constraint} \wedge \mathsf{ask}\,[\varphi] \quad \rightsquigarrow \quad \left\{ \begin{array}{c} \textit{„Subsumtion“} \\ \textit{„Weder noch“} \\ \textit{„Dissubsumtion“} \end{array} \right\}.$$

Auch hier wollen wir wieder inkrementelles Verhalten der Maschine: Nach Abarbeiten einer $\mathsf{ask}\,[\cdot]$–Eingabe und Ausgabe der entsprechenden Meldungen, soll sich die Maschine im gleichen Zustand wie vor der Ausführung befinden.

Damit haben wir zwei Punkte, die in die Maschine zu integrieren sind:

- Die Maschine muß auf Subsumtion testen.

- Die Maschine muß in der Lage sein, nach Testen der Subsumtion oder beim Fehlschlag des Tests ihren ursprünglichen Zustand wieder herzustellen.

Wir werden sehen, daß sich diese beiden Anforderungen simultan erfüllen lassen.

Aus Kapitel 8 wissen wir bereits, daß Testen von Erfüllbarkeit und Testen von Subsumtion in dem Constraintsystem RAT eng zusammenhängen: Beiden Aufgaben ist die Berechnung eines Normierers gemein.

Auch in der Maschine sind diese Zusammenhänge sehr eng gekoppelt. Es wird ausreichen, die Maschine um die Instruktionen

`ask` und `asked`

zu erweitern, und ein paar weitere, kleine Änderungen an der Unifikationsprozedur und Emulatorschleife vorzunehmen.

Der Übersetzer wird für die Benutzereingabe `ask` $[\varphi]$ die folgende Instruktionssequenz erzeugen:

`ask`	%	Test auf Subsumtion beginnen
$\cdots$	%	Instruktionen für φ
`asked`	%	Test auf Subsumtion abschließen

Wir wollen uns zuerst überlegen, wie die Maschine erweitert werden muß, damit sie den Zustand vor einer Abarbeitung wiederherstellen kann. Die einzigen Änderungen im Maschinenzustand, die zurückgenommen werden müssen, sind die, die bereits existierende Zellen betreffen.

Aber die einzige Änderung, die an Zellen während der Abarbeitung vorgenommen werden, ist das Binden von Variablenzellen und Strukturzellen an andere Zellen. Dieser Bindevorgang hat gerade den Effekt, daß durch Ändern des *ref*–Feldes aus einer echten Zelle eine unechte Zelle wird.

Um diese Änderung zurückzunehmen, reicht es aus, daß man sich Zeiger auf die Zellen merkt, deren *ref*–Felder verändert wurde. Es ist nicht nötig, den vorherigen Wert der *ref*–Felder zu speichern, da dieser immer gleich **nil** ist. Dazu führen wir den **Spurstapel** (engl. trail–stack) wie folgt ein:

Trail: **stack of** ↑*cell* ;

Hier und in den folgenden Programmteilen nehmen wir an, daß unsere PASCAL–ähnliche Programmiersprache den vordefinierten Datentyp **stack of** *Type* zusammen mit den üblichen Operationen **push**(*Type*, **stack of** *Type*), **pop**(*Type*, **stack of** *Type*), **empty**(**stack of** *Type*): **bool** und **emptystack**: **stack of** *Type* besitzt.

Auf diesen Stapel legt die Maschine, sobald der Referenzzeiger einer Zelle gesetzt wird, einen Zeiger auf die Zelle ab. Aber da die einzige Stelle in der Maschine, wo dieses Überschreiben stattfindet, in der Prozedur `unify` ist, brauchen wir auch nur diese zu ändern.

Wir werden jedoch aus Gründen, die später explizit werden, das Überschreiben von *ref*–Feldern in Variablen- und Strukturzellen voneinander trennen. Für den Fall einer Variablenzelle wird eine neue Prozedur `bind` in der Maschine vorgesehen. Damit ergibt sich die neue Unifikationsprozedur wie in an Abbildung 10.1 angegeben, wobei die Änderungen in der Prozedur grau unterlegt sind.

Durch diese Konstruktion sind wir also in der Lage, den Maschinenzustand wiederherzustellen. Doch wir werden sehen, daß der Spurstapel auch die Information enthält, die die Maschine braucht um Subsumtion oder Dissubsumtion zu entscheiden.

Wenn wir noch einmal Satz 8.22 betrachten, so liegt Subsumtion eines Normierers bezüglich einer Determinante in RAT dann vor, wenn der Normierer keinen Konflikt auslöst und die Variablen des Normierers gerade bereits durch die Determinante bestimmt sind.

In der Maschine haben wir eine analoge Situation: Nur wenn die Abarbeitung des Constraints φ in `ask` $[\varphi]$ keine neuen Bindungen von Variablenzellen herstellt, ist der Constraint φ subsumiert. Das heißt die Maschine kann Subsumtion einfach dadurch feststellen, daß nach Abarbeitung der Instruktionssequenz für φ die Instruktion `asked` den Spurstapel daraufhin untersucht, ob er einen Eintrag einer Variablenzelle enthält.

In Abbildung 10.1 wird die Boolesche Variable `Entailed` auf **`false`** gesetzt, wenn das Binden einer Variablenzelle auf dem Spurstapel registriert wird. In unserem jetzigen Szenario spielt es keine Rolle, ob das Vermerken, daß keine Subsumtion vorliegt, jetzt stattfindet, oder erst durch das Untersuchen des Spurstapelinhalts bei Abarbeitung der `asked`–Instruktion. In weiteren Ausbaustufen der Maschine wird es sich aber herausstellen, daß diese Alternative die einfachere ist.

Wird die Bindung einer Variablenzelle auf dem Spurstapel vermerkt, steht fest, daß keine Subsumtion vorliegen kann. Falsch wäre es, auf der Stelle eine Meldung „Weder noch“ auszudrucken, und die Ausführung abzubrechen. Denn in der weiteren Ausführung des Constraints kann noch ein Fehlschlag der Maschine auftreten, das heißt, es gilt Dissubsumtion.

Man beachte, daß das Ablegen auf dem Spurstapel und das Setzen der Booleschen Variable `Entailed` nur dann stattfindet, wenn die Boolesche Variable `Ask` gesetzt ist. Dadurch wird verhindert, daß die Maschine im Normalbetrieb jede Bindung auf dem Spurstapel registriert.

Der Fall eines Konflikts überträgt sich natürlich dadurch, daß bei der Unifikation die Marke *failure* angesprungen wird.

Die Operation, die die in den Spurstapel eingetragenen Änderungen wieder zurücknimmt, ist in Abbildung 10.2 gezeigt. Die Funktionsweise ist einfach: In jeder auf dem Spurstapel vermerkten Variablenzelle wird die vorgenommene Bindung wieder rückgängig gemacht.

In Abbildung 10.3 ist die um die beiden Befehle `ask` und `asked` erweiterte Emulatorschleife der Maschine dargestellt.

Die Instruktion `ask` hat nur einfache Initialisierungsaufgaben wahrzunehmen, die den Spurstapel und das damit zusammenhängende Testen auf Subsumtion betreffen. Zusätzlich wird auch eine Boolesche Variable `Ask` gesetzt, die angibt, daß sich die Maschine im “Anfrage”–Modus zur Bestimmung von Subsumtion befindet.

Diese Variable `Ask` ist wichtig, wenn bei der Unifikation ein Konflikt auftritt, also die

```
procedure bind(X, Y: ↑cell)
   X↑.ref := Y;
   if Ask
   then push(X, Trail);
        Entailed := false
   fi
end bind;

procedure unify(X, Y: ↑cell)
   deref(X); deref(Y);
   if X ≠ Y
   then if X↑.tag = var
        then bind(X, Y)
        else if Y↑.tag = var
             then bind(Y, X)
             else if X↑.sym = Y↑.sym ∧ X↑.arity = Y↑.arity
                  then push(X, Trail);
                       X↑.ref := Y;
                       for I ← 1 to X↑.arity do
                          unify(X↑.child[I], Y↑.child[I])
                       od
                  else goto failure
                  fi
             fi
        fi
   fi
end unify;
```

Abbildung 10.1: Die erweiterte Unifikationsprozedur

```
procedure undo()
   while ¬empty(Trail) do
      pop(X, Trail);
      X↑.ref := nil
   od
end undo;
```

Abbildung 10.2: Abbauen des Spurstapels

```
var Trail    : stack of ↑cell ;
    Ask      : bool := false;
    Entailed: bool;

emulate : case Code[PC] of
              ⟨ask⟩   : Trail := emptystack;
                        Ask := true;
                        Entailed := true;
              ⟨asked⟩: Ask := false;
                        if Entailed
                        then return ``Subsumiert''
                        else undo();
                             return ``Weder noch''
                        fi
                      ...

failure : if Ask
          then Ask := false;
               undo();
               return ``Dissubsumiert''
          else return ``Ausführung fehlgeschlagen''
          fi;
```

Abbildung 10.3: Emulatorschleife mit ask $[\varphi]$–Erweiterung

Marke *failure* angesprungen wird. In dem Fall, daß sich die Maschine im Anfrage–Modus befindet, wird die entsprechende Meldung ausgegeben, und der Maschinenzustand restauriert.

Auch die asked–Instruktion ist in der offensichtlichen Art und Weise implementiert. Sie sorgt dafür, daß in den normalen Modus der Maschine zurückgeschaltet wird, und daß wenn keine Subsumtion vorliegt, der ursprüngliche Maschinenzustand wiederhergestellt wird.

Man beachte, daß wenn Subsumtion vorliegt, die auf dem Spurstapel verzeichneten Änderungen von Strukturzellen nicht wieder rückgängig gemacht werden. Dieses ist deshalb nicht nötig, da es den in der Maschine repräsentierten Constraint nicht ändert.

10.2 Subsumtion in RAT mit Existenzquantoren

Im letzten Abschnitt haben wir die Maschine dahingehend erweitert, so daß sie in der Lage ist, Subsumtion in RAT zu testen. Wir werden die Maschine jetzt so erweitern, daß auch Existenzquantoren in den auf Subsumtion zu testenden Constraints erlaubt sind.

```
type cell = record
     case tag of
        var    : ref   : ↑cell ;
                local : bool;
        struct: ···;
     esac
    end;

procedure new_var(var X: ↑cell ; L: bool)
   new(X);
   X↑ := ⟨tag: var, ref: nil, local: L⟩
end new_var;
```

Abbildung 10.4: Erweiterung um lokale Variablenzellen

Dabei handelt es sich um eine für die Praxis sehr wichtige Erweiterung. Will man zum Beispiel feststellen, ob die Variable Xs an eine Liste gebunden ist, so kann man mit Hilfe von existentieller Quantifizierung die Eingabe

$$\mathsf{ask}\,[\exists \mathsf{X} \exists \mathsf{Xr}\; \mathsf{Xs} \doteq \mathsf{cons}(\mathsf{X}, \mathsf{Xr})]$$

an die Maschine richten. Ohne existentielle Quantifizerung ist das nicht möglich. Denn die Anfrage

$$\mathsf{ask}\,[\mathsf{Xs} \doteq \mathsf{cons}(\mathsf{X}, \mathsf{Xr})]$$

beinhaltet zusätzlich, ob X und Xr gleich den beiden Argumenten von cons ist.

In Kapitel 8 wurde der Begriff eines $\overline{x}$–orientierten Normierers eingeführt, um zu entscheiden, ob Subsumtion des Constraint $\exists \overline{x}\, \varphi$ vorliegt. Die Forderung an einen $\overline{x}$–orientierten Normierer war, daß keine Variable, die nicht in $\overline{x}$ vorkommt, auf eine Variable aus $\overline{x}$ abgebildet werden darf. Dementsprechend werden wir die Binde–Prozedur der Maschine umschreiben.

Die Variablen $\overline{x}$ in $\mathsf{ask}\,[\exists \overline{x}\, \varphi]$ werden wir als **lokale Variablen** bezeichnen; analog dazu werden wir nicht lokale Variablen als **globale Variablen** bezeichnen. Die Maschine wir diese lokalen Variablen genauso behandeln, wie es durch den Begriff des $\overline{x}$–orientierten Normierers vorgeschlagen wird. Das beinhaltet die folgenden beiden Punkte:

- Wenn eine lokale Variablenzelle an eine andere Zelle gebunden wird, so wird diese Bindung nicht auf dem Spurstapel vermerkt.

- Sollen zwei Variablenzellen miteinander unifiziert werden, so muß, wenn es sich um eine lokale und ein globale handelt, die lokale an die globale gebunden werden.

Die erste Erweiterung hängt natürlich jetzt damit zusammen, die Datenstruktur *cell* der Maschine um lokale Variablen zu erweitern. Dies geschieht dadurch, daß wir für eine

```
emulate: case Code[PC] of
           ⟨put_var, I⟩      : new_var(X[I], false);
           ⟨put_local_var, I⟩: new_var(X[I], true);
                               ...;
         esac;
         ...;
```

Abbildung 10.5: Emulatorschleife der Maschine

```
procedure bind(X, Y: ↑cell)
   if Y↑.tag = var ∧ Y↑.local
   then Y↑.ref := X
   else if ¬X↑.local ∧ Ask
        then push(X, Trail);
             Entailed := false
        fi;
        X↑.ref := Y
   fi
end bind;
```

Abbildung 10.6: Erweiterung des Bindens

Variablenzelle `Cell` einfach ein zusätzliches Feld *local* einführen, das angibt, ob eine Variable lokal (`Cell.`*local* = **true**) oder global (`Cell.`*local* = **false**) ist.

Die erweiterte Datentypdeklaration ist in Abbildung 10.4 dargestellt, wobei geänderte Teile grau unterlegt sind. Aus der Abbildung ist auch ersichtlich, daß die Operation `new_var` analog erweitert ist.

Zudem muß die Maschine jetzt auch die Möglichkeit haben, lokale Variablen durch eine Instruktion zu erzeugen. Dazu führen wir, analog zu der `put_var`–Instruktion, die `put_local_var`–Instruktion ein. Der Übersetzer wird jetzt für jede globale Variable zum Erzeugen wie bisher die `put_var`–Instruktion benutzen, für lokale Variablen jedoch die neu eingeführte Instruktion.

Die Integration dieser Instruktion in die Emulatorschleife ist in Abbildung 10.5 dargestellt. Man beachte, daß wegen der Erweiterung der `new_var`–Operation auch eine Änderung der `put_var`–Instruktion nötig ist.

Der letzte Punkt der Änderung der Maschine besteht jetzt darin, die `bind`–Operation zu ändern, die während der Unifikation das Binden von Variablenzellen an andere Zellen vornimmt. Wie diese zu ändern ist, haben wir bereits weiter oben überlegt. Die Umsetzung ist dann in Abbildung 10.6 gezeigt.

10.3 Integration von Konditionalen

Wir werden die Maschine jetzt so erweitern, daß sie den Kalkül $K(\mathrm{RAT})$ implementiert.

Dem Benutzer der Maschine ist es jetzt erlaubt, neben normalen Constraints φ auch Konditionale der Form $\langle\varphi;\sigma;\tau\rangle$ einzugeben. Wir werden jetzt zur Vereinfachung davon ausgehen, daß es dem Benutzer nicht mehr erlaubt ist, Ausdrücke der Form $\mathsf{ask}\,[\varphi]$ einzugeben. Dies ist sinnvoll, da $\mathsf{ask}\,[\cdot]$ nur eingeführt wurde, um das Testen von Subsumtion getrennt von den Implementierungsproblemen für Konditionale zu betrachten.

Zu Erinnerung: Die operationale Semantik eines Konditionales $\langle\varphi;\sigma;\tau\rangle$ war durch die beiden folgenden Reduktionsregeln definiert:

$$\varphi \wedge \langle\psi;\sigma;\tau\rangle \longrightarrow \varphi \wedge \left\{ \begin{array}{ll} \sigma & \text{falls } \varphi \models_{\mathrm{RAT}} \psi \\ \tau & \text{falls } \varphi \models_{\mathrm{RAT}} \neg\psi \end{array} \right\}.$$

Im weiteren werden wir σ den **then–Teil** und τ den **else–Teil** des Konditionals nennen.

10.3.1 Suspension und Aufwecken eines Konditionals

Wie die Maschine Subsumtion und Dissubsumtion feststellt, wissen wir ja bereits. Bis jetzt genügte es im Fall, daß weder Subsumtion noch Dissubsumtion vorlag, die Halde wiederherzustellen und im Grundzustand der Maschine weiterzumachen.

Nun ist die Situation anders: Nachdem die Maschine feststellt, daß weder Subsumtion noch Dissubsumtion eines Wächters φ vorliegt, muß das Konditional **suspendieren**: Suspension eines Konditionals bedeutet nichts anderes, als daß die Abarbeitung des Konditionals unterbrochen wird und erst dann wieder aufgenommen wird, wenn sich eine Änderung des Maschinenzustands ergeben hat, die eventuell zu Subsumtion oder Dissubsumtion des schon berechneten Wächters führen kann.

Wann der Wächter eines Konditionals erneut auf Subsumtion oder Dissubsumtion untersucht werden muß, läßt sich einfach überlegen. Suspension liegt dann vor, wenn der Wächter des Konditionals erfolgreich abgearbeitet wurde, jedoch auf dem Spurstapel Bindungen von globalen Variablenzellen realisiert wurden. Nur wenn diese Variablenzellen gebunden werden, kann Subsumtion oder Dissubsumtion eintreten. Wir sagen auch, daß **Konditional suspendiere auf eine Variable**. Damit haben wir das Kriterium, wann ein Konditional erneut betrachtet werden muß: Wenn eine Variable, auf die es suspendiert hat, gebunden wird. Dann sagen wir auch, daß das Konditional **geweckt** werden muß.

10.3.2 Das Skript

Wird ein Konditional wieder geweckt, so muß erneut auf Subsumtion oder Dissubsumtion getestet werden. Eine Methode wäre, alle Maschineninstruktionen, die den Wächter implementieren, erneut auszuführen: Das ist aber nicht sehr effizient.

Wir werden hier eine andere Technik verwenden, die nur die wesentlichen Dinge erneut ausführt. Wird der Wächter eines Konditionals ausgeführt, so werden alle Bindungen von Zellen an andere Zellen auf dem Spurstapel registriert. Diese Änderungen, die die

Abarbeitung des Wächters in den Maschinenzustand einbringt, können wir also dazu benutzen, um den Wächter eines Konditionales in effizienter Art und Weise auszunutzen.

Wie man sich leicht überlegt, ist nur die Information über Bindungen von Variablenzellen relevant. Die Bindung von Strukturzellen dient nur dazu, bei der Abarbeitung die Terminierung zuzusichern. Die Information, die aus Paaren von Verweisen auf Zellen besteht, die aneinander gebunden waren, werden wir als **Skript** bezeichnen.

Da eine Wiederausführung des Wächters nur stattfindet, wenn sich die Bindung mindestens einer globalen Variablen geändert hat, reicht es jedoch nicht, alle auf dem Spurstapel verzeichneten Bindungen wieder direkt in die Maschine zu übernehmen. Anstatt dessen wird die Wiederausführung des Wächters dadurch stattfinden, daß die Variablenzelle und die Zelle, die nach der Information des Spurstapels daran gebunden war, miteinander unifiziert werden.

10.4 Die K(RAT)–Maschine

Nachdem wir die zentralen Operationen zur Implementierung des Kalküls K(RAT) herausgearbeitet gaben, können wir diese in die Maschine umsetzen.

10.4.1 Neue Instruktionen

Zuerst werden wir angeben, auf welche Instruktionssequenz der Übersetzer ein Konditional

$$\langle\varphi;\sigma;\tau\rangle \wedge \mu$$

abbildet.

Genau wie für die Übersetzung von `ask` $[\varphi]$ sehen wir eine Instruktion `cond` in der Maschine vor, die anzeigt, daß hier die Instruktionssequenz für den Wächter eines Konditionals beginnt. Arbeitet die Maschine den Wächter ab, sagen wir auch, daß sie sich im **Testzustand** befindet. Sonst befindet sie sich im **Grundzustand**.

Dann muß eine Instruktion eingeführt werden, die anzeigt, daß der Wächter zu Ende ist, und der Test auf Subsumtion und Dissubsumtion vorgenommen werden muß: Diese wollen wir mit `then` bezeichen. Ist der Wächter subsumiert, wird die Ausführung einfach nach der `then`–Instruktion fortgesetzt. Ist er dissubsumiert, so muß ein Sprung zu der Instruktionssequenz für τ durchgeführt werden, die Adresse dieser Instruktionssequenz muß also der `cond`–Instruktion als Parameter mitgegebene werden. Genauso für den Fall, daß der Wächter suspendiert: Dann muß bei der Instruktionssequenz für μ fortgesetzt werden.

Damit haben wir die nötigen Maschineninstruktionen in der folgenden Art und Weise:

```
type cond = record
     script : stack of ⟨↑cell ,↑cell⟩;
     then   : nat;
     else   : nat;
     dead   : bool;
    end;

var Cond: ↑cond
```

Abbildung 10.7: Repräsentation von Konditionalen

```
       cond  Else, Cont    %  Start des Konditionals
       ···                 %  Instruktionen für φ
       then                %  Ende des Wächters
       ···                 %  Instruktionen für σ
       end                 %  Ende von σ
Else:  ···                 %  Instruktionen für τ
       end                 %  Ende von τ
Cont:  ···                 %  Instruktionen für μ
```

Die Instruktionssequenzen für σ und τ werden mit der `end`–Instruktion abgeschlossen. Diese Instruktion wird für eine Fortsetzung der Abarbeitung in der Maschine sorgen, die Details werden wir jedoch auf später verschieben.

10.4.2 Repräsentation suspendierter Konditionale

In dem Fall, daß ein Konditional suspendiert werden muß, muß die ganze zu einem Konditional gehörige Information gespeichert werden, und so bald das Konditional geweckt wird, muß diese Information wieder verfügbar gemacht werden.

Dazu führen wir den Datentyp *cond* für Konditionale ein. Gleichzeitig sehen wir eine neue globale Variable `Cond` in der Maschine vor, die auf das gerade bearbeitete Konditional zeigt, oder, falls gerade kein Konditional abgearbeitet wird, gleich **nil** ist. Die Definition des Datentyps ist in Abbildung 10.7 gezeigt.

Das Feld *script* in dem Verbund wird gerade die Codierung des Skripts aufnehmen. Es ist als Stapel von Paaren zwischen Verweisen auf Zellen dargestellt. Aus der vorherigen Argumentation ist auch der Zweck der Felder *then* und *else* bekannt: Sie werden die Adresse der Instruktion hinter der `then`–Instruktion, respektive die Adresse für den else–Teil des Konditionals enthalten.

Zusätzlich enthält der Verbund noch das Feld *dead*. Dieses Feld gibt an, ob der Wächter des Konditionals nach dem Aufwecken noch einmal abgearbeitet werden muß, oder ob dieses nicht mehr nötig ist. Wieso diese Information wichtig ist, wird uns das nächste Beispiel zeigen.

Beispiel 10.1 (Suspension auf mehrere Variablen) Sei das folgende Konditional ge-

```
loop : if ¬empty(Jobs)
       then pop(J, Jobs);
            case J of
              ⟨jump, N⟩: PC := N;
                         goto emulate;
              ⟨retry, C⟩ if C↑.dead
                         then goto loop
                         else Cond := C;
                              goto retry
                         fi

            esac
       else return ''done''
       fi
```

Abbildung 10.8: Abarbeitung von Aufträgen

geben:

$$\langle \mathsf{X} \doteq \mathsf{a} \wedge \mathsf{Y} \doteq \mathsf{a}; \mathsf{Z} \doteq \mathsf{eq}; \mathsf{Z} \doteq \mathsf{neq} \rangle \,.$$

Wenn das Konditional bezüglich des leeren Constraints $\top$ abgearbeitet wird, so suspendiert das Konditional sowohl auf die Variable X als auch auf die Variable Y. Anders formuliert heißt das: Wenn X oder Y gebunden wird, muß das Konditional geweckt werden.

Nehmen wir jetzt an, daß der Constraint $\mathsf{X} \doteq \mathsf{b}$ hinzugefügt wird. Dann wird das Konditional geweckt, der Wächter ist dissubsumiert, also kann das ganze Konditional simplifiziert werden.

Kommt jetzt später noch der Constraint $\mathsf{Y} \doteq \mathsf{a}$ hinzu, so muß verhindert werden, daß die Maschine versucht, das Konditional noch einmal zu simplifizieren. Um dieses zu verhindern, wird die Maschine nur solche Konditionale `C` erneut abarbeiten, wenn `C↑.`*dead*`=`**`false`** ist. □

10.4.3 Der Auftragsstapel

Jetzt müssen wir überlegen, wie die Abarbeitung in der Maschine prinzipiell geregelt wird. Dazu führen wir einen **Auftragsstapel** ein, auf den die Maschine **Aufträge** ablegen wird, so daß die Maschine später an Hand dieser Aufträge bestimmte Arbeiten erledigen kann. Wir werden zwei verschiedene Art von Aufträgen vorsehen:

- Der Auftrag ⟨`retry, C`⟩ verlangt, daß das suspendierte Konditional `C↑` erneut abgearbeitet wird.
- ⟨`jump, N`⟩ verlangt, daß die Maschine einen Sprung zur Adresse `N` durchführt.

```
type cell = record
     case tag of
        var   : ref   : ↑cell ;
                local : bool;
                sleep : stack of ↑cond ;
        struct: ···;
     esac
    end;

procedure new_var(var X: ↑cell ; L: bool)
   new(X);
   X↑ := ⟨tag: var, ref: nil, local: L, sleep: emptystack⟩
end new_var;
```

Abbildung 10.9: Erweiterung der Datentypen für RAT–Constraints

Dazu führen wir in der Maschine eine globale Variable `Jobs` ein, dies ist also gerade der Auftragsstapel. Die Maschine wird jetzt also solange arbeiten, bis sich nichts mehr auf dem Auftragsstapel befindet.

Wie diese Abarbeitung der Aufträge dann konkret implementiert ist, ist in Abbildung 10.8 dargestellt. Das angegebene Fragment ist im wesentlichen selbsterklärend. Man kann jetzt erkennen, daß die Maschine erst dann die Abarbeitung vollkommmen abgeschlossen hat, wenn auf dem Auftragsstapel nichts mehr vorhanden ist.

10.4.4 Aufwecken von Konditionalen

Wir haben bereits festgestellt, daß Konditionale dann aufgeweckt werden müssen, wenn Variablen, auf die sie suspendiert haben, gebunden werden. Um diese Funktionalität in die Maschine zu integrieren, müssen wir Variablenzellen so erweitern, daß es möglich ist in einer Variablenzelle zu vermerken, daß Konditionale bei Bindung der Zelle geweckt werden müssen.

In Abbildung 10.9 ist dargestellt, wie die Deklaration einer Variablenzelle für die Maschine aussieht. Jede Variablenzelle bekommt ein zusätzliches Feld *sleep*, das auf einen Stapel von suspendierten Konditionalen zeigt.

Wenn nun die Operation `bind` eine globale Variablenzelle bindet, und die Maschine sich im Grundzustand befindet, so werden alle in der Variablenzelle in *sleep* als schlafend gekennzeichneten Konditionale dadurch geweckt, daß für jedes dieser Konditionale `C` ein Auftrag ⟨`retry,C`⟩ auf den Auftragsstapel legt.

Die Fallunterscheidung, ob die Maschine einen Wächter ausführt, oder nicht, vermeidet einerseits unnötiges Aufwecken von Konditionalen, andererseits wird dadurch auch verhindert, daß unnötige Einträge auf dem Spurstapel vorgenommen werden. Genau ausgeführt ist dies in Abbildung 10.10.

```
procedure bind(X, Y: ↑cell)
   if Y↑.tag = var ∧ Y↑.local
   then Y↑.ref := X
   else if ¬X↑.local
        then if Cond = nil
             then while ¬empty(X↑.sleep) do
                     pop(C, X↑.sleep);
                     push(⟨retry, C⟩, Jobs)
                  od
             else push(X, Trail);
                  Entailed := false;
             fi
        fi;
        X↑.ref := Y
   fi
end bind;
```

Abbildung 10.10: Aufwecken beim Binden

Man beachte, daß das Verhindern von unnötigem Aufwecken für die Terminierung der Maschine essentiell ist. Dazu betrachten wir das folgende Beispiel.

Beispiel 10.2 Für dieses Beispiel nehmen wir an, daß das Wecken von Konditionalen auch dann durchgeführt wird, wenn sich die Maschine im Konditionalmodus befindet. Wenn man die beiden Konditionale

$$\langle \mathsf{X} \doteq \mathsf{Y}; \top; \bot \rangle \wedge \langle \mathsf{X} \doteq \mathsf{Y}; \top; \bot \rangle$$

in der Maschine abarbeitet, wenn der kanonische Constraint in der Maschine $\mathsf{X} \doteq \mathsf{Y}$ weder subsumiert noch dissubsumiert, passiert das folgende:

- Nach Abarbeiten des Wächters des ersten Konditionals wird das Konditional suspendiert, und in den Variablenzellen für X und Y wird eingetragen, daß dieses Konditional beim Binden von X oder Y geweckt werden muß.

- Bei der Abarbeitung des zweiten Konditionals werden zwei Weckaufträge für das erste Konditional auf den Auftragsstapel gelegt, da die Variablen X und Y bei der Abarbeitung des Wächters gebunden werden. Aber auch dieses Konditional suspendiert.

- Die Abarbeitung wird bei dem jetzt aufgeweckten ersten Konditional fortgesetzt. Das weckt dann wieder das zweite Konditional, und so weiter. . . .

□

```
procedure mk_script(C: ↑cond)
   while ¬empty(Trail) do
      pop(X, Trail);
      case X↑.tag of
         struct: X↑.ref := nil;
         var   : push(⟨X, X↑.ref⟩, C↑.script);
                 push(C, X↑.sleep);
                 if X↑.ref↑.tag= var
                 then push(C, X↑.ref↑.sleep)
                 fi
                 X↑.ref := nil
            esac
   od
end mk_script;

procedure do_script(C: ↑cond)
   while ¬empty(C↑.script) do
      pop(⟨X, Y⟩, C↑.script);
      unify(X, Y)
   od;
end do_script;
```

Abbildung 10.11: Erzeugen und Abarbeiten eines Skripts

10.4.5 Herstellen und Ausführen des Skripts

In Abschnitt 10.3.2 haben wir diskutiert, daß die Information nach der Ausführung eines Wächters in ein Skript gesichert werden kann, und daß bei erneuter Berechnung des Wächters dieses Skript dann abgearbeitet werden muß.

Die beiden Operationen, einmal das Herstellen eines Skripts aus dem Spurstapel und das Wiederabarbeiten eines Skripts lassen sich leicht formulieren. Gezeigt sind die beiden Operationen in Abbildung 10.11.

Dabei entspricht die Operation `mk_script` dem Erzeugen eines Skripts aus dem Spurstapel. Man beachte, daß diese Prozedur als Parameter einen Konditionalknoten entgegennimmt, und ihre Einträge in das diesem Knoten zugeordnete Skript vornimmt.

Gleichzeitig mit dem Eintragen der gebundenen Paare in das Skript wird in die Variablenzellen, auf die das Konditional suspendiert, eingetragen, daß dieses Konditional geweckt werden muß. Man beachte, daß wenn zwei Variablenzellen aneinander gebunden waren, das Konditional auch in die Variablenzelle eingetragen werden muß, deren Bindung nicht auf dem Spurstapel registriert wurde.

Die duale Prozedur `do_script` ist die direkte Umsetzung der bereits weiter oben beschriebenen Wiederausführung des Wächters.

10.4.6 Die Emulatorschleife

Hier wollen wir die Emulatorschleife der K(RAT)–Maschine angeben und näher erläutern. Im wesentlichen werden nur noch ein paar nähere Details zur Implementierung der neuen Instruktionen angeben.

Die Emulatorschleife zusammen mit weiteren Programmteilen zur Verwaltung von Konditionalen ist in Abbildung 10.12 angegeben.

Trifft die Maschine auf die `cond`–Instruktion, so wird der Datenblock für ein neues Konditional angelegt. Man beachte, daß noch keine Adresse in das *then*–Feld eingetragen wird. Dieses wird erst dann vorgenommen, wenn die Maschine nach Abarbeitung des Wächters des Konditionals auf den `then`–Befehl trifft.

Wie schon bei der Implementierung der `ask`–Instruktion wird für das Testen von Subsumtion der Spurstapel initialisiert und festgelegt, daß bisher noch keine Bindung zwischen globalen Variablen vorliegt, also daß der Wächter bis jetzt subsumiert ist.

Die `then`–Instruktion hat jetzt festzustellen, ob Subsumtion oder Suspension vorliegt. Liegt Subsumtion vor, wird die Abarbeitung einfach bei der nächsten Instruktion fortgesetzt. Man beachte, daß es nicht nötig ist, das Konditional als tot zu markieren (durch `Cond↑.`*dead* := **`true`**), da dieses Konditional das erste Mal bearbeitet wurde, es also keine Möglichkeiten gibt, es noch einmal aufzuwecken.

Zusätzlich müssen wir uns auch davon überzeugen, daß dieses Vorgehen mit den eventuell vorhandenen lokalen Variablen des Wächters verträglich ist. Es darf nicht eintreten, daß globale Variablen an lokale Variablen gebunden sind: Dann müßten die lokalen Variablen zu globalen geändert werden. Aber das kann auch nicht eintreten: Die Prozedur `bind` sorgt dafür, daß keine globalen an lokale Variablen gebunden werden.

Liegt keine Subsumtion vor, so muß das Konditional suspendiert werden. Dazu wird zuerst die Fortsetzungsadresse in das *then*–Feld des Konditionals eingetragen, damit nach eventuellem erfolgreichen Wiederaufwecken die Abarbeitung an der richtigen Stelle fortgestzt werden kann.

Die Aktionen, die nötig sind, um ein Konditional zu suspendieren, werden durch einen Sprung zur Marke *suspend* eingeleitet. Hier wird wie bereits bekannt, daß Skript nach den Informationen auf dem Spurstapel berechnet, und in den Konditionalknoten eingetragen. Insbesondere wird das Konditional auch in den Weckstapel aller Variablen eingetragen, auf die das Konditional suspendiert (siehe Abbildung 10.11).

Die Wiederabarbeitung von einem Wächter eines Konditionals (diese geschieht dadurch, daß ein entsprechender Auftrag vom Auftragsstapel abgearbeitet wird), wird durch das Abarbeiten des Skripts des Konditionals durchgeführt. Liegt bei dieser Abarbeitung dann Subsumtion vor, wird die Abarbeitung des Konditional abgebrochen, das Konditional als tot markiert, und die Abarbeitung beim then–Teil des Konditionals fortgesetzt.

Zuletzt bleibt noch zu überlegen, wie Fehlschlag in der Maschine gehandhabt wird. In dem Fall, wenn sich die Maschine nicht in der Abarbeitung eines Wächter befindet, bricht sie nach Ausgabe einer Meldung die Abarbeitung ab.

Sonst liegt Dissubsumtion des gerade bearbeiteten Wächters vor. Also muß der Maschinenzustand zurückgesetzt werden. Dies geschieht wie üblich durch die Prozedur `undo`

```
emulate : case Code[PC] of
            ⟨cond, Else, Cont⟩: new(Cond);
                                Cond↑.else := Else;
                                Cond↑.dead := false;
                                Trail := emptystack;
                                Entailed := true;
                                push(⟨jump, Cont⟩, Jobs);
            ⟨then⟩            : if Entailed
                                then Cond := nil
                                else Cond↑.then := PC + 1;
                                     goto suspend
                                fi;
            ⟨end⟩             : goto loop ;
                              ...
          esac;

suspend : mk_script(Cond);
          Cond  := nil;
          goto loop ;

retry :   Entailed := true
          do_script(Cond);
          if Entailed
          then PC := Cond↑.then ;
               Cond↑.dead := true;
               Cond := nil;
               goto emulate
          else goto suspend
          fi;

failure : if Cond = nil
          then return ''failure''
          else undo();
               PC := Cond↑.else ;
               Cond↑.dead := true;
               Cond := nil;
               goto emulate
          fi
```

Abbildung 10.12: Emulatorschleife für die K(RAT)–Maschine (siehe Abbildung 10.2). Auch hier muß dann die Abarbeitung des Konditionals beendet, das Konditional als tot markiert werden und die Abarbeitung des else-Teils des Konditionals gestartet werden.

10.5 Änderung des Bindungsbereiches

In den letzten beiden Abschnitten sind wir davon ausgegangen, daß sich der Bindungsbereich eines Existenzquantors im Wächter eines Konditionals nur über den Wächter erstreckt.

Wenn man aber den Wächter dazu benutzt, festzustellen, ob ein Term eine spezielle Bauart hat, und den if–Teil des Konditionals dazu, die Unterterme dieses Terms zu manipulieren, ist der bisherige Bindungsbereich hinderlich. Sinnvoller wäre es, wenn sich die existentielle Quantifizierung auch über den then–Teil erstrecken würde, das Konditional also die folgende logische Semantik hätte:

$$\langle \exists\overline{x}\, \varphi; \sigma; \tau\rangle \models\!\mid \exists\overline{x}(\varphi \wedge \sigma) \vee (\neg\exists\overline{x}\, \sigma \wedge \tau).$$

Das nächste Beispiel unterstreicht unsere Argumentation.

Beispiel 10.3 Ist Xs eine Liste, dann soll das erste Element der Liste gleich Y gesetzt werden. Dazu müßen wir das folgende Konditional

$$\langle \exists \mathsf{X} \exists \mathsf{Xr}\, \mathsf{Xs} \doteq \mathsf{cons}(\mathsf{X},\mathsf{Xr}); \exists \mathsf{X} \exists \mathsf{Xr}\, (\mathsf{Xs} \doteq \mathsf{cons}(\mathsf{X},\mathsf{Xr}) \wedge \mathsf{X} \doteq \mathsf{Y})\, ; \bot\rangle$$

hinschreiben. Mit dem veränderten Bindungsbereich kann man kürzer

$$\langle \exists \mathsf{X} \exists \mathsf{Xr}\, \mathsf{Xs} \doteq \mathsf{cons}(\mathsf{X},\mathsf{Xr}); \mathsf{X} \doteq \mathsf{Y}; \bot\rangle$$

schreiben. □

Die erste Schreibweise ist nicht nur umständlich, sondern auch ineffizient: Bei der Abarbeitung des then–Teils des Konditionals muß erneut dieselbe Unifikation wie im Wächter unternommen werden.

Wir wollen jetzt eine Erweiterung der Maschine angeben, die diesen erweiterten Bindungsbereich der Existenzquantoren realisiert. Eine einfache Lösung wäre es, nach erfolgreicher Abarbeitung des Wächters alle lokalen Variablen des Wächters durch Ausführen einer Instruktion `make_global X`i global zu machen. Das ist aber immer noch ineffizient, denn für n im Wächter quantifizierte Variablen hat dieses Vorgehen eine Komplexität von $O(n)$.

Was wir anstreben ist eine Lösung, die die Komplexität $O(1)$ hat. Die Idee ist, daß wir die Information, ob Variablen lokal oder global sind, dem Konditional zuordnen, in dem sie quantifiziert sind. Wenn des Wächter des Konditionals subsumiert ist, wird die Maschine die Information dahingehend ändern, daß alle Variablen ab jetzt als global zu betrachten sind.

Technisch integrieren wir das dadurch, daß wir den Datenblock eines Konditionals um ein Feld *home* erweitern, das einen Zeiger auf eine Boolesche Variable enthält. Skizziert ist dies in Abbildung 10.13.

```
type cond = record
       ...
       home : ↑bool;
    end;
```

Abbildung 10.13: Erweiterung des Konditionals

```
emulate : case Code[PC] of
            ⟨cond, Else, Cont⟩: new(Cond);
                                new(Cond↑.home);
                                Cond↑.home↑ := true;
                                ...;
            ⟨then⟩            : if Entailed
                                then Cond↑.home↑ := false;
                                     Cond := nil
                                else ...
                                fi;
            ⟨put_local_var, I⟩ : new_var(X[I], Cond↑.home);
                               ...


retry :   ...;
          if Entailed
          then Cond↑.home↑ := false;
               ...
          else goto suspend
          fi;
```

Abbildung 10.14: Implementierung des erweiterten Bindungsbereichs

Gleichzeitig ändert sich jetzt natürlich auch die Definition des Datentyps *cell*: Das Feld *local* bekommt den neuen Datentyp ↑**bool**. Wie sich jetzt die Initialisierung einer Variablenzelle mit der Prozedur `new_var` und der Zugriff auf die Lokalitätsinformation einer Variablenzelle (dies geschieht in der Prozedur `bind`) ändert, ist einfach, die durchzuführenden Änderungen seien dem Leser überlassen.

Wird jetzt das Konditional `Cond` bei Abarbeitung des `cond`–Befehls initialisiert, wird eine neue Speicherzelle erzeugt, auf die das Feld `Cond↑.home` zeigt. Diese Speicherzelle wird mit **true** initialisiert. Das heißt: Die Lokalitätsinformation aller Variablen, die mit der Instruktion `put_local_var` Xi bei der Abarbeitung dieses Wächters erzeugt werden, bekommen automatisch eine Referenz auf diese Speicherzelle. Zu sehen sind diese beiden Aspekte in Abbildung 10.14.

Nach erfolgreicher Subsumtion des Wächters, wird jetzt einfach der Wert der Speicherzelle, auf die das Feld `Cond`$\uparrow$`.`*home* zeigt auf **`false`** gesetzt. Damit werden alle lokalen Variablen dieses Wächters global. Zu sehen ist dieses in Abbildung 10.14, einmal bei der Implementierung der `put_local_var X`i–Instruktion, andererseits bei dem auf die Sprungmarke *retry* folgenden Programmteil, der den Fall der Subsumtion eines Wächters nach Wiederabarbeitung implementiert.

Teil IV

Semantik

Kapitel 11

Semantik definiter Programme

In diesem Kapitel wollen wir eine denotationale Semantik für eine Klasse von logischen Programmen angeben. Insbesondere werden wir die Modelle dieser Programme näher untersuchen, und die Existenz von ausgezeichneten Modellen beweisen. Mehr noch: Wir werden zeigen, daß ein kanonisches Modell in einem gewissen Sinne berechnet werden kann.

Bevor wir uns jedoch dieser Untersuchung zuwenden, werden wir uns einen nützlichen technischen Apparat zurechtlegen.

11.1 Verbände

Zur Erinnerung: Eine **partielle Ordnung** ist ein Paar $(M, \leq)$, wobei $\leq$ eine binäre Relation auf der Menge M ist, die reflexiv ($\forall m \in M : \ \ m \leq m$), antisymmetrisch ($\forall m, n \in M : \ \ m \leq n$ und $n \leq m \ \Rightarrow \ n = m$) und transitiv ($\forall m, n, o \in M : \ \ n \leq m$ und $m \leq o \ \Rightarrow \ n \leq o$) ist.

Sei $(M, \leq)$ eine partielle Ordnung, und N eine Teilmenge von M. Dann heißt $m \in M$ eine **obere Schranke von** N, falls:

$$\forall n \in N : \quad n \leq m.$$

Analog dazu heißt $m \in M$ eine **untere Schranke von** N, falls:

$$\forall n \in N : \quad m \leq n.$$

$m \in M$ heißt **kleinste obere Schranke von** N oder **Supremum von** N, falls m eine obere Schranke ist, und es gilt:

$$\forall n \in N : \qquad n \text{ ist obere Schranke von } N \ \Rightarrow \ m \leq n.$$

Analog dazu ist die **größte untere Schranke von** N oder das **Infimum von** N definiert. Die kleinste obere Schranke einer Menge N wird mit $\bigsqcup N$, die größte untere Schranke einer Menge N mit $\bigsqcap N$ bezeichnet.

Definition 11.1 (Verband) *Eine partielle Ordnung $(M, \leq)$ heißt* **Verband**, *falls für alle $m, n \in M$ sowohl $\bigsqcap\{m, n\}$ als auch $\bigsqcup\{m, n\}$ existiert.*

Ein Verband $(M, \leq)$ heißt vollständig, wenn für jede Teilmenge $N \subseteq M$ sowohl $\bigsqcap N$ als auch $\bigsqcup N$ existieren.

Proposition 11.1 *Suprema und Infima bezüglich einer partiellen Ordnung sind eindeutig bestimmt, falls sie existieren.*

Die obige Proposition rechtfertigt die folgende Schreibweise, die wir im weiteren verwenden werden. Für zwei Elemente m und n eines Verbandes $(M, \leq)$ schreiben wir Infimum und Supremum in Infixschreibweise, also:

$$\begin{aligned} m \sqcup n &:= \bigsqcup\{m, n\} \\ m \sqcap n &:= \bigsqcap\{m, n\}. \end{aligned}$$

Insbesondere hat ein vollständiger Verband ein kleinstes und ein größtes Element, die wir im folgenden mit $\perp$ und $\top$ bezeichnen wollen.

Beispiel 11.1 (Verbände)

1. Sei M eine beliebige, nichtleere Menge. Dann ist der **Potenzmengenverband** $(\mathcal{P}(M), \subseteq)$ ein vollständiger Verband, wobei $\mathcal{P}(M)$ die Potenzmenge von M bezeichnet. Weiterhin gilt für jedes $N \subseteq \mathcal{P}(()\, M)$:

   $$\begin{aligned} \bigsqcup N &= \bigcup N \\ \bigsqcap N &= \bigcap N. \end{aligned}$$

   Die kleinsten und größten Elemente sind $\emptyset$ und M.

2. Sei $(M, \leq)$ ein Verband und M endlich. Dann ist $(M, \leq)$ vollständig.

3. Bezeichne $\mathbb{R}$ die Menge der reellen Zahlen und $\leq$ die übliche Ordnungsrelation auf den reellen Zahlen. Dann ist $(\mathbb{R}, \leq)$ ein nicht vollständiger Verband. Weiterhin kann dieser Verband durch Hinzunehmen von Elementen $\{-\infty, \infty\}$ in der üblichen Art und Weise vollständig gemacht werden.

□

11.2 Fixpunktsätze für vollständige Verbände

Wir nennen eine Teilmenge $D \subseteq M$ eines vollständigen Verbandes $(M, \leq)$ **gerichtet**, falls für jede endliche Teilmenge $F \subseteq D$ eine obere Schranke in D existiert.

Definition 11.2 *Seien $(M, \leq)$ und $(N, \leq)$ vollständige Verbände und $T : M \to N$ eine Abbildung.*

1. *T heißt* **monoton**, *falls für alle $m_1, m_2 \in M$ gilt:*
$$m_1 \leq m_2 \Rightarrow T(m_1) \leq T(m_2).$$

2. *T heißt* **stetig**, *falls für jede gerichtete Teilmenge $D \subseteq M$ gilt:*
$$T(\bigsqcup D) = \bigsqcup T(D).$$

Proposition 11.2 *Seien $(M, \leq)$ und $(N, \leq)$ vollständige Verbände und $T : M \to N$ eine stetige Abbildung. Dann ist T monoton.*

Beweis Seien $m_1, m_2 \in M$ mit $m_1 \leq m_2$. Dann ist die endliche Menge $\{m_1, m_2\}$ gerichtet, da $m_2 = m_1 \sqcup m_2$. Also gilt

$$\begin{aligned} T(m_2) &= T(m_1 \sqcup m_2) \\ &= T(m_1) \sqcup T(m_2) \quad (T \text{ stetig}) \\ &\geq T(m_1). \end{aligned}$$

□

Definition 11.3 *Sei $(M, \leq)$ ein vollständiger Verband und $T : M \to M$ eine Abbildung. Dann heißt $m \in M$*

1. **Fixpunkt von** T, *falls $T(m) = m$.*
2. **Präfixpunkt von** T, *falls $T(m) \leq m$.*
3. **Postfixpunkt von** T, *falls $m \leq T(m)$.*

Satz 11.3 (Tarski [Tar55]) *Sei $(M, \leq)$ ein vollständiger Verband und $T : M \to M$ eine monotone Abbildung. Dann hat T einen kleinsten und einen größten Fixpunkt. Dabei ist*
$$s := \bigsqcap \{m \in M \mid T(m) \leq m\}$$
der kleinste Fixpunkt von T, und
$$g := \bigsqcup \{m \in M \mid m \leq T(m)\}$$
der größte Fixpunkt von T.

Beweis Wir zeigen, daß g der größte Fixpunkt von T ist. Dazu bezeichnen wir mit G die Menge $\{m \in M \mid m \leq T(m)\}$ aller Postfixpunkte von T.

1. Wir zeigen, daß $g \in G$: Für alle $m \in G$ gilt nach der Defintion von g, daß $m \leq g$. Da T monoton ist, gilt
$$\forall m \in G : \quad m \leq T(m) \leq T(g),$$
das heißt also, daß $T(g)$ eine obere Schranke von G ist. Damit gilt $g = \bigsqcup G \leq T(g)$, also ist $g \in G$.

2. Wir zeigen nun $T(g) \leq g$: Wegen 1. haben wir, daß $g \leq T(g)$. Wenden wir auf beide Seiten T, haben wir auf Grund der Montonie von T:
$$T(g) \leq T(T(g)).$$
Also ist $T(g) \in G$, damit aber auch $T(g) \leq \bigsqcup G = g$.
3. Aus 1. und 2. folgt, daß $T(g) = g$. Also ist g Fixpunkt von T. Jeder weitere Fixpunkt g' von T ist in G. Damit ist $g' \leq \sqcup G = g$, also ist g größter Fixpunkt von T.

Dual dazu zeigt man: s ist kleinster Fixpunkt von T. □

Proposition 11.4 *Sei $(M, \leq)$ ein vollständiger Verband und $T : M \to M$ eine monotone Abbildung. Dann gilt:*
$$\perp \leq T(\perp) \leq T^2(\perp) \leq T^3(\perp) \leq \cdots \qquad .$$

Beweis Da $\perp$ kleinstes Element ist, gilt $\perp \leq T(\perp)$. Mit vollständiger Induktion über k zeigt man leicht, daß $T^k(\perp) \leq T^{k+1}(\perp)$. □

Wenn $(N_i)_{i \in I}$ eine Familie von Elementen eines vollständigen Verbandes ist, verwenden wir im weiteren auch die folgenden Abkürzungen:
$$\bigsqcup_{i \in I} N_i := \bigsqcup \{N_i |\ i \in I\} \quad \text{und} \quad \bigsqcap_{i \in I} N_i := \bigsqcap \{N_i |\ i \in I\}\,.$$

Satz 11.5 (Kleene [Kle52]) *Sei $(M, \leq)$ ein vollständiger Verband und $T : M \to M$ eine stetige Abbildung. Dann ist*
$$\mathit{fix} := \bigsqcup_{k \geq 0} T^k(\perp)$$
der kleinste Fixpunkt von T.

Beweis Zuerst zeigen wir, daß *fix* ein Fixpunkt ist. Wegen der Propositionen 11.2 und 11.4 ist $\{T^k(\perp) \mid k \geq 0\}$ gerichtet. Da $\perp \leq T(\perp)$, gilt zusätzlich
$$\mathit{fix} = \bigsqcup_{k \geq 1} T^k(\perp) \tag{11.1}$$

Also haben wir:
$$\begin{aligned} T(\mathit{fix}) &= T\left(\bigsqcup_{k \geq 0} T^k(\perp)\right) \\ &= \bigsqcup_{k \geq 0} T\left(T^k(\perp)\right) && (T \text{ stetig}) \\ &= \bigsqcup_{k \geq 1} T^k(\perp) \\ &= \mathit{fix} && (\text{wegen } (11.1)) \end{aligned}$$

Jetzt zeigen wir, daß *fix* der kleinste Fixpunkt von T ist. Dazu sei m ein beliebiger Fixpunkt von T. Wir zeigen mit Induktion über k:

$$T^k(\bot) \leq m. \tag{11.2}$$

Es ist $T^0(\bot) = \bot \leq m$. Unter der Induktionsannahme $T^k(\bot) \leq m$ gilt wegen der Monotonie von T:

$$T^{k+1}(\bot) = T(T^k(\bot)) \leq T(m) = m.$$

Damit ist (11.2) gezeigt.

Also ist m eine obere Schranke von $\{T^k(\bot) \mid k \geq 0\}$, das heißt

$$\mathit{fix} = \bigsqcup \{T^k(\bot) \mid k \geq 0\} \leq m.$$

$\square$

11.3 Definite Programme

In diesem Abschnitt wollen wir die Menge von Programmen angeben, deren Semantik wir im weiteren untersuchen wollen. Wir verallgemeinern den Ansatz aus dem ersten Teil des Skripts dahingehend, daß wir jetzt keine Programme mehr betrachten, die nur über der Herbrand Struktur $\mathcal{H}$ rechnen, sondern solche, die sich auf ein beliebiges Constraintsystem beziehen.

Außerdem werden wir keine Hornklauseln mehr betrachten, sondern gleich Formelmengen, die denen der Clarkschen Vervollständigung von Hornklauselprogrammen entsprechen. Nur, wie oben schon erwähnt, werden wir beliebige Theorien anstatt der Theorie CET zulassen.

Dazu gehen wir im folgenden von einem Constraintsystem $\Gamma = (\Sigma, \Delta, \mathrm{CON})$ aus. Zur Erinnerung: Σ war die Signatur des Constraintsystems, Δ eine konsistente Theorie über Σ und CON die Menge der Constraints.

Zusätzlich führen wir jetzt noch eine Menge R von Relationssymbolen ein, für die $R \parallel \Sigma$ gilt. Diese Relationssymbole wollen wir auch **Prozeduren** nennen.

Damit kommen wir zu der folgenden Definition:

$$\begin{array}{lcl} \varphi,\, \psi & : & \Gamma\text{–}\mathit{Constraints} \\ A,\, B & ::= & r(x_1, \ldots, x_n) \\ \sigma,\, \tau,\, \mu,\, \nu & ::= & \varphi \quad \mid \quad A \quad \mid \quad \sigma \wedge \tau \quad \mid \quad \sigma \vee \tau \quad \mid \quad \exists x\, \sigma \end{array}$$

Elemente von $[\sigma]$ nennen wir auch **definite Formeln**.

Ein **definites Programm** Π ist wie folgt definiert: Π ist eine Menge von Formeln $r\overline{x} \leftrightarrow \sigma$ mit:

1. $\overline{x}$ ist linear, das heißt alle Variablen in $\overline{x}$ sind paarweise verschieden.
2. $\mathcal{V}(\sigma) \subseteq \mathcal{V}(\overline{x})$.

3. Für jedes $r \in R$ enthält Π genau eine Äquivalenz $r\overline{x} \leftrightarrow \sigma$.

Im weitern wird es unser Ziel, die denotationale Semantik dieser Programme zu untersuchen. Worauf wir dabei aus sind, ist, die Modelle eines definiten Programmes Π zu untersuchen.

11.4 Algebraische Programme

Im folgenden bezeichnen wir die Menge der definiten Formeln bezüglich eines Constraintsystems $\Gamma = (\Sigma, \Delta, \mathrm{CON})$ und einer Menge von Relationszeichen R mit $\mathrm{DFOR}(\Sigma \uplus R)$.

Definition 11.4 (Algebraische Programme) *Ein* **algebraisches Programm** *ist eine Abbildung* $\Pi : \mathrm{DFOR}(\Sigma \uplus R) \to \mathrm{DFOR}(\Sigma \uplus R)$ *mit den folgenden Eigenschaften:*

1. $\Pi(\varphi) = \varphi$.
2. $\Pi(\sigma \wedge \tau) = \Pi(\sigma) \wedge \Pi(\tau)$.
3. $\Pi(\sigma \vee \tau) = \Pi(\sigma) \vee \Pi(\tau)$.
4. $\Pi(\exists x\, \sigma) = \exists x\, \Pi(\sigma)$.
5. $\mathcal{V}(\Pi(\sigma)) \subseteq \mathcal{V}(\sigma)$.
6. *Für alle A und für alle Substitutionen θ, die Variablen auf Variablen abbilden, existiert eine α–Variante σ von $\Pi(A)$ mit θ frei für σ, so daß $\theta\sigma$ eine α–Variante von $\Pi(\theta A)$ ist.*

Die letzte Eigenschaft in der obigen Definition kann man folgendermassen skizzieren, wobei $\equiv_\alpha$ die Relation der α–Umbennenung bezeichnet:

$$
\begin{array}{ccccc}
A & \xrightarrow{\Pi} & \Pi(A) & \equiv_\alpha & \sigma \\
\theta\big\downarrow & & & & \big\downarrow\theta \\
\theta A & \xrightarrow{\Pi} & \Pi(\theta A) & \equiv_\alpha & \theta\sigma
\end{array}
$$

Der Zusammenhang zwischen logischen und algebraischen Programmen ist dann noch einmal in der folgenden Proposition formuliert.

Proposition 11.6 *Zu jedem logischen Programm Π_L existiert ein algebraisches Programm Π, so daß*

$$\forall A \leftrightarrow \sigma \in \Pi_L : \quad \Pi(A) = \sigma.$$

Die obige Proposition rechtfertigt also, daß wir uns im folgenden nur noch mit algebraischen Programmen beschäftigen.

11.5 Von algebraischen Programmen zu Strukturen

Unser Ziel wird es jetzt sein, von den algebraischen Programmen zu geeigneten Strukturen überzugehen, uns somit von jeglicher Syntax zu befreien.

Definition 11.5 (Modell eines algebraischen Programms) *Eine $\Sigma \uplus R$-Struktur ist ein* **Modell des algebraischen Programms** Π *oder* $\mathcal{A} \models \Pi$, *wenn*

$$\forall \sigma : \quad \mathcal{A}\,[\![\sigma]\!] = \mathcal{A}\,[\![\Pi(\sigma)]\!]$$

gilt.

In der obigen Definition kommt es auf die Denotation beliebiger definiter Formeln an. Diese Eigenschaft werden wir jetzt dadurch einfacher handhabbar machen, in dem wir eine möglichst kleine Menge von Formeln auswählen, die die Denotation eines algebraischen Programms festlegt. Dazu führen wir den folgenden Begriff ein.

Eine Menge Ω von linearen Atomen heißt **Atombasis** (oder kürzer **Basis**), wenn sie für jede Prozedur $r \in R$ genau ein Atom enthält. Insbesondere bilden somit die linken Seiten eines definiten Programmes eine Basis.

Die folgende Proposition sagt gerade aus, daß der Begriff Basis gerechtfertigt ist.

Proposition 11.7 *Sei Ω eine Atombasis. Dann gilt:*

$$\mathcal{A} \models \Pi \iff \forall A \in \Omega : \quad \mathcal{A} \models A \leftrightarrow \Pi(A).$$

Beweis Nach Definition 11.5 müssen wir

$$\forall \sigma : \quad \mathcal{A}\,[\![\sigma]\!] = \mathcal{A}\,[\![\Pi(\sigma)]\!] \tag{11.3}$$

zeigen. Dieses tun wir mit Induktion über den Aufbau von σ.

1. $\sigma = \varphi$: Gilt, da $\Pi(\varphi) = \varphi$.
2. $\sigma = A$: Nach Definition der Semantik von $\leftrightarrow$ ist $\mathcal{A} \models A \leftrightarrow \Pi(A)$ äquivalent zu (11.3).
3. $\sigma = \mu \wedge \nu$: Hier haben wir:

$$\begin{array}{rcll} \mathcal{A}\,[\![\mu \wedge \nu]\!] & = & \mathcal{A}\,[\![\mu]\!] \cap \mathcal{A}\,[\![\nu]\!] & \\ & = & \mathcal{A}\,[\![\Pi(\mu)]\!] \cap \mathcal{A}\,[\![\Pi(\nu)]\!] & \text{(Induktionsvoraussetzung)} \\ & = & \mathcal{A}\,[\![\Pi(\mu) \wedge \Pi(\nu)]\!] & \\ & = & \mathcal{A}\,[\![\Pi\,(\mu \wedge \nu)]\!] & \end{array}$$

Die restlichen beiden Fälle bleiben dem Leser überlassen. □

Im weiteren werden wir eine „mengentheoretische Version“ des Existenzquantors benötigen. Dazu führen wir die folgende Notation ein.

Definition 11.6 *Sei $\mathcal{A}$ eine Struktur und x eine Variable. Dann ist die Abbildung $\Pi_x^{\mathcal{A}} : \mathcal{P}(VAL(\mathcal{A})) \to \mathcal{P}(VAL(\mathcal{A}))$ wie folgt definiert:*

$$\Pi_x^{\mathcal{A}}(V) := \{\alpha[x \leftarrow a] \mid \alpha \in V, a \in \mathcal{U}(\mathcal{A})\}.$$

Daß diese Abbildung das gewünschte leistet, sagt uns gerade die nächste Proposition.

Proposition 11.8 *Sei $\mathcal{A}$ eine $\Sigma \uplus R$–Struktur. Dann gilt für jede Formel σ:*

$$\mathcal{A}\,[\![\exists x\, \sigma]\!] = \Pi_x^{\mathcal{A}}(\mathcal{A}\,[\![\sigma]\!]).$$

Beweis Durch einfaches Nachrechnen erhalten wir:

$$\begin{aligned} \mathcal{A}\,[\![\exists x\, \sigma]\!] &= \{\alpha[x \leftarrow a] \mid \alpha \in \mathcal{A}\,[\![\sigma]\!], a \in \mathcal{U}(\mathcal{A})\} \\ &= \Pi_x^{\mathcal{A}}(\mathcal{A}\,[\![\sigma]\!]) \end{aligned}$$

□

Sei $\mathcal{A}$ eine Struktur und $V \subseteq VAL(\mathcal{A})$. Wir sagen, daß V die Variable x **einschränkt**, falls $V \neq \Pi_x^{\mathcal{A}}(V)$. Die **Menge der eingeschränkten Variablen** $B(V)$ ist wie folgt definiert:

$$B(V) := \{x \in VAR \mid V \text{ schränkt } x \text{ ein}\}.$$

Die nächste Proposition sagt gerade, daß eine Formel höchstens ihre freien Variablen einschränkt.

Proposition 11.9 *Sei $\mathcal{A}$ eine Struktur und sei σ eine beliebige Formel. Dann gilt:*

$$B(\mathcal{A}\,[\![\sigma]\!]) \subseteq \mathcal{V}(\sigma).$$

Beweis Übungsaufgabe! □

Sei $\mathcal{A}$ eine $\Sigma \uplus R$–Struktur. Dann bezeichnen wir mit $\mathcal{A}_\perp$ die Struktur, die mit $\mathcal{A}$ bis auf die Definition von $\mathcal{A}\,[\![r]\!]$ für $r \in R$ übereinstimmt. Für alle $r \in R$ ist $\mathcal{A}_\perp\,[\![r]\!]$ als $\emptyset$ definiert.

Analog dazu bezeichnen wir mit $\mathcal{A}_\top$ die Struktur, für die für jede n-stellige Prozedur $r \in R$ $\mathcal{A}\,[\![r]\!] = VAL(\mathcal{A})^n$ ist.

Proposition 11.10 *Sei $\mathcal{A}$ eine $\Sigma \uplus R$–Struktur, Ω eine Basis und $\omega : \Omega \to \mathcal{P}(VAL(\mathcal{A}))$ eine Abbildung mit:*

$$\forall A \in \Omega : \quad B(\omega(A)) \subseteq \mathcal{V}(A).$$

Dann existiert genau eine Struktur $\mathcal{B}$ mit:

1. $\mathcal{A}_\perp = \mathcal{B}_\perp$.
2. $\forall A \in \Omega : \quad \mathcal{B}\,[\![A]\!] = \omega(A)$.

Beweis Übungsaufgabe! □

Jetzt sind wir in der Lage, einem algebraischen Programm eine eindeutig bestimmte Struktur zuzuordnen. Im weiteren sei eine Atombasis Ω festgelegt.

Proposition 11.11 *Sei Π ein algebraisches Programm. Für jede $\Sigma \uplus R$–Struktur $\mathcal{A}$ gibt es eine eindeutig bestimmte Struktur $\mathcal{B}$, so daß:*

1. $\mathcal{A}_\perp = \mathcal{B}_\perp$.
2. $\forall A \in \Omega : \quad \mathcal{B}\,[\![A]\!] = \mathcal{A}\,[\![\Pi(A)]\!]$.

Beweis Die Aussage dieser Proposition folgt direkt aus der Proposition 11.10, wenn wir die Abbildung $\omega : \Omega \to \mathcal{P}\,(\mathit{VAL}(\mathcal{A}))$ wie folgt definieren:

$$\omega(A) := \mathcal{A}\,[\![\Pi(A)]\!]\,.$$

□

Von einem algebraischen Programm ausgehend, können wir jetzt eine eindeutige Abbildung auf Strukturen definieren. Dazu bezeichnen wir die Menge aller $\Sigma \uplus R$–Strukturen mit $\mathit{STRUC}(\Sigma \uplus R)$.

Definition 11.7 *Die Abbildung $\hat{\Pi} : \mathit{STRUC}(\Sigma \uplus R) \to \mathit{STRUC}(\Sigma \uplus R)$ ordnet einer Struktur $\mathcal{A} \in \mathit{STRUC}(\Sigma \uplus R)$ die nach Proposition 11.11 eindeutig definierte Struktur $\hat{\Pi}(\mathcal{A})$ zu.*

Die Abbildung $\hat{\Pi}$ wollen wir auch wieder einfach mit Π bezeichen. Das nächste Lemma beschreibt jetzt für die Denotation definiter Formeln den Zusammenhang zwischen dem algebraischen Programm und der Abbildung auf Strukturen.

Lemma 11.12 *Für alle definiten Formeln σ gilt:*

$$(\Pi(\mathcal{A}))\,[\![\sigma]\!] = \mathcal{A}\,[\![\Pi(\sigma)]\!]\,.$$

Beweis Beweis mit Induktion über den Aufbau von σ.

1. $\sigma = \varphi$: Für Constraints φ gilt:
$$(\Pi(\mathcal{A}))\,[\![\varphi]\!] = \mathcal{A}\,[\![\varphi]\!] = \mathcal{A}\,[\![\Pi(\varphi)]\!]\,.$$
2. $\sigma = A$: Gilt direkt nach Proposition 11.11.
3. $\sigma = \mu \wedge \nu$: Es gilt:

$$\begin{aligned}
(\Pi(\mathcal{A}))\,[\![\mu \wedge \nu]\!] &= (\Pi(\mathcal{A}))\,[\![\mu]\!] \cap (\Pi(\mathcal{A}))\,[\![\nu]\!] \\
&= \mathcal{A}\,[\![\Pi(\mu)]\!] \cap \mathcal{A}\,[\![\Pi(\nu)]\!] \qquad \text{(Induktionsvoraussetzung)} \\
&= \mathcal{A}\,[\![\Pi(\mu) \wedge \Pi(\nu)]\!] \\
&= \mathcal{A}\,[\![\Pi\,(\mu \wedge \nu)]\!]
\end{aligned}$$

Die restlichen Fälle seien dem Leser überlassen. □

11.6 Minimale und maximale Modelle

In diesem Abschnitt werden wir die Modelle eines definiten Programmes angeben. Den Zusammenhang zwischen diesen Modellen und der Abbildung $\Pi : STRUC(\Sigma \uplus R) \to STRUC(\Sigma \uplus R)$ gibt uns die nächste Proposition.

Proposition 11.13 *$\mathcal{A}$ ist genau dann ein Modell des algebraischen Programms Π, wenn $\mathcal{A}$ ein Fixpunkt von $\Pi : STRUC(\Sigma \uplus R) \to STRUC(\Sigma \uplus R)$ ist, das heißt:*

$$\mathcal{A} \models \Pi \quad \Longleftrightarrow \quad \Pi(\mathcal{A}) = \mathcal{A}.$$

Beweis Einzusehen ist die obige Aussage mit der folgenden Argumentation:

$$\begin{aligned} \Pi(\mathcal{A}) = \mathcal{A} \quad &\Longleftrightarrow \quad \forall A \in \Omega : \quad (\Pi(\mathcal{A}))\,[\![A]\!] = \mathcal{A}\,[\![A]\!] \\ &\Longleftrightarrow \quad \forall A \in \Omega : \quad \mathcal{A}\,[\![\Pi(A)]\!] = \mathcal{A}\,[\![A]\!] \qquad \text{(nach Lemma 11.12)} \end{aligned}$$

□

Die obige Proposition gibt uns gerade an, wie die weitere Argumentation laufen wird. Modelle können wir als Fixpunkte der Abbildung Π erhalten. Also werden wir einen vollständigen Verband angeben, so daß wir die Fixpunktsätze für vollständige Verbände ausnutzen können.

Dazu werden wir $\Sigma \uplus R$–Strukturen in der folgenden Art und Weise ordnen. Seien $\mathcal{A}, \mathcal{B} \in STRUC(\Sigma \uplus R)$:

$$\mathcal{A} \leq \mathcal{B} \quad :\Longleftrightarrow \quad \mathcal{A}_\perp = \mathcal{B}_\perp \text{ und } \forall r \in R: \ \mathcal{A}\,[\![r]\!] \subseteq \mathcal{B}\,[\![r]\!]\,.$$

Diese Ordnungsrelation wird wiederum schon eindeutig durch die Denotation der Elemente einer Atombasis festgelegt. Genauer gilt die folgende Proposition.

Proposition 11.14 *Seien $\mathcal{A}, \mathcal{B} \in STRUC(\Sigma \uplus R)$ mit $\mathcal{A}_\perp = \mathcal{B}_\perp$. Dann gilt*

$$\mathcal{A} \leq \mathcal{B} \quad \Longleftrightarrow \quad \forall A \in \Omega : \ \mathcal{A}\,[\![A]\!] \subseteq \mathcal{B}\,[\![A]\!]\,.$$

Beweis Übungsaufgabe! □

In der nächsten Proposition werden wir den vollständigen Verband angeben, an dem wir interessiert sind.

Proposition 11.15 *Sei $\mathcal{A}$ eine $\Sigma \uplus R$–Struktur. Dann ist*

$$(\{\mathcal{B} \in STRUC(\Sigma \uplus R) |\ \mathcal{A}_\perp = \mathcal{B}_\perp\}\,, \leq)$$

ein vollständiger Verband. Seien die Strukturen $(\mathcal{B}_i)_{i \in I}$ Elemente diese Verbandes. Dann gilt für alle $r \in R$:

$$\left(\bigsqcup_{i \in I} \mathcal{B}_i\right)[\![r]\!] = \bigcup_{i \in I} \mathcal{B}_i\,[\![r]\!] \quad \text{und} \quad \left(\bigsqcap_{i \in I} \mathcal{B}_i\right)[\![r]\!] = \bigcap_{i \in I} \mathcal{B}_i\,[\![r]\!]\,.$$

Beweis Übungsaufgabe! □

Die Ordnung von Strukturen überträgt sich jetzt in naheliegender Weise auf die Inklusion von Denotationen beliebiger Formeln bezüglich dieser Strukturen.

Proposition 11.16 *Seien $\mathcal{A}, \mathcal{B} \in STRUC(\Sigma \cup R)$. Dann gilt:*

$$\mathcal{A} \leq \mathcal{B} \;\Rightarrow\; \forall \sigma : \; \mathcal{A}\,[\![\sigma]\!] \subseteq \mathcal{B}\,[\![\sigma]\!] \,.$$

Beweis Wir führen den Beweis mit Induktion über den Aufbau von σ. Bevor wir jedoch den eigentlichen Beweis beginnen, überlegen wir uns, daß Durchschnitt und Vereinigung von Mengen selber monoton ist. Das heißt, für Mengenfamilien $(M_i)_{i\in I}, (N_i)_{i\in I}$ gilt:

$$\forall i \in I : \; M_i \subseteq N_i \quad \Rightarrow \quad \bigcap_{i\in I} M_i \subseteq \bigcap_{i \in I} N_i \tag{11.4}$$

und

$$\forall i \in I : \; M_i \subseteq N_i \quad \Rightarrow \quad \bigcup_{i\in I} M_i \subseteq \bigcup_{i \in I} N_i . \tag{11.5}$$

1. $\sigma = \varphi$: Dieser Fall folgt aus der Tatsache, daß $\mathcal{A}\,[\![\varphi]\!] = \mathcal{B}\,[\![\varphi]\!]$ gilt.
2. $\sigma = A$: Folgt aus der Proposition 11.14.
3. $\sigma = \mu \wedge \nu$: Man sieht leicht ein, daß:

$$\begin{array}{lcll} \mathcal{A}\,[\![\mu \wedge \nu]\!] & = & \mathcal{A}\,[\![\mu]\!] \cap \mathcal{A}\,[\![\nu]\!] & \\ & \subseteq & \mathcal{B}\,[\![\mu]\!] \cap \mathcal{B}\,[\![\nu]\!] & \text{(Induktionsvoraussetzung und (11.4))} \\ & = & \mathcal{B}\,[\![\mu \wedge \nu]\!] . & \end{array}$$

4. $\sigma = \mu \vee \nu$: Analog zu dem obigen Fall unter Anwendung von (11.5) anstatt (11.4).
5. $\sigma = \exists x\, \mu$: Hier gilt:

$$\begin{array}{lcll} \mathcal{A}\,[\![\exists x\, \mu]\!] & = & \Pi_x^{\mathcal{A}}\,(\mathcal{A}\,[\![\mu]\!]) & \\ & = & \displaystyle\bigcup_{\alpha \in \mathcal{A}[\![\mu]\!]} \{\alpha[x \leftarrow a] \mid a \in \mathcal{U}\,(\mathcal{A})\} & \\ & \subseteq & \displaystyle\bigcup_{\alpha \in \mathcal{B}[\![\mu]\!]} \{\alpha[x \leftarrow b] \mid b \in \mathcal{U}\,(\mathcal{B})\} & \text{(Induktionsvor. und (11.5))} \\ & = & \Pi_x^{\mathcal{B}}\,(\mathcal{B}\,[\![\mu]\!]) & \\ & = & \mathcal{B}\,[\![\exists x\, \mu]\!] . & \end{array}$$

□

Aus der obigen Proposition folgt auch direkt die wichtige Eigenschaft der Abbildung $\Pi : STRUC(\Sigma \uplus R) \rightarrow STRUC(\Sigma \uplus R)$, monoton zu sein.

Proposition 11.17 (Monotonie von Π) *Die Abbildung $\Pi : STRUC(\Sigma \uplus R) \to STRUC(\Sigma \uplus R)$ ist monoton, das heißt:*

$$\forall \mathcal{A}, \mathcal{B} \in STRUC(\Sigma \uplus R) : \quad \mathcal{A} \leq \mathcal{B} \;\Rightarrow\; \Pi(\mathcal{A}) \leq \Pi(\mathcal{B}).$$

Beweis Sei also $\mathcal{A} \leq \mathcal{B}$ und $A \in \Omega$. Nach Proposition 11.14 reicht es, daß folgende zu zeigen:

$$\begin{array}{rcll} (\Pi(\mathcal{A}))\,[\![A]\!] & = & \mathcal{A}\,[\![\Pi(A)]\!] & \text{(Lemma 11.12)} \\ & \subseteq & \mathcal{B}\,[\![\Pi(A)]\!] & \text{(Proposition 11.16)} \\ & = & (\Pi(\mathcal{B}))\,[\![A]\!] & \text{(Lemma 11.12).} \end{array}$$

□

Damit sind bei unserem ersten Ziel, die Existenz von Modellen zu zeigen angelangt. Genauer gilt der folgende Satz.

Satz 11.18 (Minimales und maximales Modell von Π) *Sei Π ein algebraisches Programm und $\mathcal{A}$ eine $\Sigma \uplus R$–Struktur. Dann hat*

$$\{\mathcal{B} \mid \mathcal{A}_\perp = \mathcal{B}_\perp \text{ und } \mathcal{B} \models \Pi\} \tag{11.6}$$

ein kleinstes und ein größtes Element.

Beweis Nach Proposition 11.15 ist $(\{\mathcal{B} \in STRUC(\Sigma \uplus R) \mid \mathcal{A}_\perp = \mathcal{B}_\perp\}, \leq)$ ein Verband und nach Proposition 11.17 ist die Abbildung Π auf diesem Verband monoton. Damit folgt mit dem Fixpunktsatz von Tarski (Satz 11.3), daß Π einen kleinsten und größten Fixpunkt hat.

Nach Proposition 11.13 hat also (11.6) ein kleinstes und größtes Element. □

11.7 Darstellung des minimalen Modells

Neben der Existenzaussage für ein minimales Modell der Abbildung Π wollen wir auch noch angeben, wie es sich berechnen läßt. Wir werden das Hauptergebnis über den Fixpunktsatz von Kleene (Satz 11.5) erhalten. Dazu brauchen wir die Stetigkeit der Abbildung Π.

Lemma 11.19 *Sei $(\mathcal{A}_i)_{i \in I}$ eine gerichtete Familie von $\Sigma \uplus R$–Strukturen. Dann gilt:*

$$\left(\bigsqcup_{i \in I} \mathcal{A}_i\right)[\![\sigma]\!] = \bigcup_{i \in I} \mathcal{A}_i\,[\![\sigma]\!].$$

Beweis Durch strukturelle Induktion über σ. Wir bezeichnen mit $\mathcal{A}$ die Struktur $\mathcal{A}_{i\perp}$ für ein beliebiges $i \in I$, da sowieso alle Strukturen $\mathcal{A}_{i\perp}$ gleich sind.

1. $\sigma = \varphi$: Da $\mathcal{A}_i [\![\varphi]\!] = \mathcal{A}_j [\![\varphi]\!]$ für alle $i, j \in I$ gilt, haben wir für ein beliebiges $i \in I$:
$$\left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\varphi]\!] = \mathcal{A}_i [\![\varphi]\!] = \bigcup_{i\in I} \mathcal{A}_i [\![\varphi]\!] .$$

2. $\sigma = A$: Dieser folgt aus der Proposition 11.15.

3. $\sigma = \exists x\, \mu$:
$$\begin{aligned}
\left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\exists x\, \mu]\!] &= \Pi_x^{\mathcal{A}} \left(\left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\mu]\!]\right) \\
&= \Pi_x^{\mathcal{A}} \left(\bigcup_{i\in I} \mathcal{A}_i [\![\mu]\!]\right) \qquad \text{(Induktionsvor.)} \\
&= \bigcup_{i\in I} \Pi_x^{\mathcal{A}} (\mathcal{A}_i [\![\mu]\!]) \\
&= \bigcup_{i\in I} \mathcal{A}_i [\![\exists x\, \mu]\!] .
\end{aligned}$$

4. $\sigma = \mu \vee \nu$:
$$\begin{aligned}
\left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\mu \vee \nu]\!] &= \left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\mu]\!] \cup \left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\nu]\!] \\
&= \bigcup_{i\in I} \mathcal{A}_i [\![\mu]\!] \cup \bigcup_{i\in I} \mathcal{A}_i [\![\nu]\!] \qquad \text{(Induktionsvor.)} \\
&= \bigcup_{i\in I} (\mathcal{A}_i [\![\mu]\!] \cup \mathcal{A}_i [\![\nu]\!]) \\
&= \bigcup_{i\in I} \mathcal{A}_i [\![\mu \vee \nu]\!] .
\end{aligned}$$

5. $\sigma = \mu \wedge \nu$: Wir zeigen erst eine Inklusionsbeziehung von rechts nach links:
$$\begin{aligned}
\left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\mu \wedge \nu]\!] &= \left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\mu]\!] \cap \left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\nu]\!] \\
&= \bigcup_{i\in I} \mathcal{A}_i [\![\mu]\!] \cap \bigcup_{i\in I} \mathcal{A}_i [\![\nu]\!] \qquad \text{(Induktionsvor.)} \\
&\supseteq \bigcup_{i\in I} (\mathcal{A}_i [\![\mu]\!] \cap \mathcal{A}_i [\![\nu]\!]) \\
&= \bigcup_{i\in I} \mathcal{A}_i [\![\mu \wedge \nu]\!] .
\end{aligned}$$

Nachzutragen bleibt jetzt noch, daß tatsächlich auch
$$\bigcup_{i\in I} \mathcal{A}_i [\![\mu]\!] \cap \bigcup_{i\in I} \mathcal{A}_i [\![\nu]\!] \subseteq \bigcup_{i\in I} (\mathcal{A}_i [\![\mu]\!] \cap \mathcal{A}_i [\![\nu]\!])$$
gilt. Sei also $\alpha \in \bigcup_{i\in I} \mathcal{A}_i [\![\mu]\!] \cap \bigcup_{i\in I} \mathcal{A}_i [\![\nu]\!]$. Also existieren $i \in I$ und $j \in I$, so daß
$$\alpha \in \mathcal{A}_i [\![\mu]\!] \cap \mathcal{A}_j [\![\nu]\!] .$$

Da $(\mathcal{A}_i)_{i\in I}$ gerichtet ist, existiert ein $k \in I$, so daß

$$\mathcal{A}_i \leq \mathcal{A}_k \quad \text{und} \quad \mathcal{A}_j \leq \mathcal{A}_k.$$

Nach Proposition 11.16 gilt dann auch:

$$\alpha \in \mathcal{A}_k [\![\mu]\!] \cap \mathcal{A}_k [\![\nu]\!] .$$

Damit haben wir dann aber auch

$$\alpha \in \mathcal{A}_k [\![\mu]\!] \cap \mathcal{A}_k [\![\nu]\!] \subseteq \bigcup_{i\in I} (\mathcal{A}_i [\![\mu]\!] \cap \mathcal{A}_i [\![\nu]\!]) .$$

□

Nach diesem technischen Lemma kommen wir zu der Aussage, daß Π tatsächlich stetig ist.

Proposition 11.20 (Stetigkeit von Π) *Die Abbildung $\Pi : STRUC(\Sigma \uplus R) \to STRUC(\Sigma \uplus R)$ ist stetig.*

Beweis Sei $(\mathcal{A}_i)_{i\in I}$ eine gerichtete Familie von $\Sigma \uplus R$–Strukturen. Nach Proposition 11.10 reicht es, die folgende Aussage zu zeigen:

$$\forall A \in \Omega : \quad \left(\Pi\left(\bigsqcup_{i\in I} \mathcal{A}_i\right)\right) [\![A]\!] = \left(\bigsqcup_{i\in I} \Pi(\mathcal{A}_i)\right) [\![A]\!] .$$

Man beachte, daß wegen der Monotonie von Π (Proposition 11.17) auch gilt:

$$(\Pi(\mathcal{A}_i))_{i\in I} \text{ ist gerichtet.} \tag{11.7}$$

Sei also $A \in \Omega$. Dann gilt:

$$\begin{aligned}
\left(\Pi\left(\bigsqcup_{i\in I} \mathcal{A}_i\right)\right) [\![A]\!] &= \left(\bigsqcup_{i\in I} \mathcal{A}_i\right) [\![\Pi(A)]\!] && \text{(Lemma 11.12)}\\
&= \bigcup_{i\in I} \mathcal{A}_i [\![\Pi(A)]\!] && \text{(Lemma 11.19)}\\
&= \bigcup_{i\in I} (\Pi(\mathcal{A}_i)) [\![A]\!] && \text{(Lemma 11.12)}\\
&= \left(\bigsqcup_{i\in I} \Pi(\mathcal{A}_i)\right) [\![A]\!] && \text{(Lemma 11.19, (11.7)).}
\end{aligned}$$

□

Damit haben wir einen zweiten wichtigen Satz für Modelle eines algebraischen Programms.

Satz 11.21 (Darstellung des minimalen Modells von Π**)** *Sei* Π *ein algebraisches Programm und* $\mathcal{A}$ *eine* $\Sigma \uplus R$*–Struktur. Dann ist*

$$\bigsqcup_{k \geq 0} \Pi^k(\mathcal{A}_\perp)$$

ein minimales Modell von Π*.*

Beweis Da Π nach Proposition 11.20 stetig ist, folgt die Behauptung mit dem Fixpunktsatz von Kleene. □

11.8 Beispiele

In Teil I hatten wir bereits Modelle der Clarkschen Vervollständigung comp(P) betrachtet. Hier wollen wir ein paar Beispiele untersuchen, die unseren in diesem Kapitel gemachten Zugang etwas beleuchten.

Ein wichtiger Begriff ist der folgende.

Definition 11.8 (Persistente Erweiterung) *Sei* $\mathcal{A}$ *eine* Σ*–Struktur. Dann heißt die* $\Sigma \uplus R$*–Struktur* $\mathcal{B}$ *eine* **persistente Erweiterung von** $\mathcal{A}$*, falls gilt:*

1. $\mathcal{U}(\mathcal{A}) = \mathcal{U}(\mathcal{B})$.
2. $\forall s \in \Sigma : \quad \mathcal{A}[\![s]\!] = \mathcal{B}[\![s]\!]$.

Für alle diese Beispiel wählen wir das Constraintsystem CET(Σ) (siehe Abschnitt 6.1 auf Seite 55). Betrachten werden wir persistente Erweiterungen des Standardmodells $\mathcal{H}$ für die Theorie CET. Die Menge der Relationssymbole R werden wir gerade jeweils als die Symbole der linken Seiten eines Programmes wählen.

Im ersten Beispiel ist das minimale und das maximale Modell voneinander verschieden.

Beispiel 11.2 Sei Π das folgende logische Programm:

$$\mathsf{r(X)} \leftrightarrow \mathsf{r(X)}.$$

Sei $\mathcal{A}$ eine $\Sigma \uplus \{\mathsf{r}\}$ Erweierung von $\mathcal{H}$. Dann ist $\mathcal{A}_\perp$ das minimale und $\mathcal{A}_\top$ das maximale Modell von Π. □

In dem nächsten Beispiel fallen minimales und maximales Modell zusammen.

Beispiel 11.3 Sei Π das folgende Programm:

$$\begin{array}{rcl}\mathsf{nat(X)} & \leftrightarrow & \mathsf{X} \doteq \mathsf{0} \\ & \vee & \exists \mathsf{Y}\,(\mathsf{X} \doteq \mathsf{s(Y)} \wedge \mathsf{nat(Y)})\end{array}$$

Dann gibt es genau eine persistente Erweiterung $\mathcal{M}$ von $\mathcal{H}$, die Modell von Π ist. Für diese Modell gilt

$$\mathcal{M}\,[\![\mathsf{nat}]\!] = \left\{ \mathsf{0}, \begin{array}{c}\mathsf{s}\\ |\\ \mathsf{0}\end{array}, \begin{array}{c}\mathsf{s}\\ |\\ \mathsf{s}\\ |\\ \mathsf{0}\end{array}, \cdots \right\}.$$

□

Das Programm in Beispiel 11.2 kann man als pathologisch ansehen. Das nächste Beispiel zeigt aber, daß auch bei „vernünftigen“ Beispielen das minimale und das maximale Programm auseinander fallen können.

Beispiel 11.4 Gegeben sei das folgende Programm Π:

$$\begin{array}{rcl}
\mathsf{arc}(\mathsf{X},\mathsf{Y}) & \leftrightarrow & \mathsf{X} \doteq \mathsf{a} \wedge \mathsf{Y} \doteq \mathsf{b} \\
& \vee & \mathsf{X} \doteq \mathsf{c} \wedge \mathsf{Y} \doteq \mathsf{d} \\
& \vee & \mathsf{X} \doteq \mathsf{d} \wedge \mathsf{Y} \doteq \mathsf{c} \\
\mathsf{reach}(\mathsf{X}) & \leftrightarrow & \mathsf{X} \doteq \mathsf{a} \\
& \vee & \exists \mathsf{Y}\,(\mathsf{reach}(\mathsf{Y}) \wedge \mathsf{arc}(\mathsf{Y},\mathsf{X}))
\end{array}$$

Die Intuition hinter dem obigen Programm ist, daß $\mathsf{reach}(\mathsf{X})$ gelten soll, wenn X von dem Knoten a erreichbar ist. Eine Skizze des durch arc spezifizierten Graphen ist in Abbildung 3.4 auf Seite 34 dargestellt.

Seien $\mathcal{S}, \mathcal{G}$ wiederum $\Sigma \uplus \{\mathsf{arc}, \mathsf{reach}\}$ persistente Erweiterungen von $\mathcal{H}$, und sei $\mathcal{S}$ das minimale und $\mathcal{G}$ das maximale Modell von Π. Dann gilt:

$$\mathcal{S}\,[\![\mathsf{arc}]\!] = \mathcal{G}\,[\![\mathsf{arc}]\!] = \{(\mathsf{a},\mathsf{b}), (\mathsf{c},\mathsf{d}), (\mathsf{d},\mathsf{c})\}$$

und

$$\begin{array}{rcl}
\mathcal{S}\,[\![\mathsf{reach}]\!] & = & \{\mathsf{a},\mathsf{b}\} \\
\mathcal{G}\,[\![\mathsf{reach}]\!] & = & \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\}.
\end{array}$$

□

Teil V

Anhänge

Anhang A

Syntax und Semantik der Prädikatenlogik

A.1 Syntax

Zuerst wollen wir uns mit der Syntax der Prädikatenlogik der ersten Stufe beschäftigen. Diese setzt sich aus einer Menge von Symbolen, der **Signatur**, und aus syntaktischen Objekten, die nach festen Vorschriften aus dieser Signatur gewonnen werden, den **Termen** und **Formeln** zusammen.

A.1.1 Signatur

Eine Signatur legt Namen und Stelligkeit (Anzahl der Argumente) für die zu betrachtenden Funktionen und Relationen fest. Sie parametrisiert somit die Menge der syntaktische konstruierbaren Terme und Formeln.

Die Basis für unsere syntaktischen Objekte sind durch zwei Symbolmengen gegeben. Einerseits die Menge FUN der **Funktionssymbole**, andererseits die Menge $PR\ddot{A}$ der **Prädikatensymbole**, wobei die beiden Mengen disjunkt sein müssen.

Die **Stelligkeitsfunktion** oder **Aritätsfunktion** $|\cdot| : \Sigma \to \mathbb{N}_0$ ordnet jedem Symbol seine **Stelligkeit** zu.

Nullstellige Funktionssymbole heißen auch **Konstanten** oder **Individuenkonstanten**, nullstellige Prädikatensymbole auch **Aussagenvariablen**.

Im weiteren nehmen wir an, daß $f, g, h \in FUN$ und $p, q, r \in PR\ddot{A}$ sind.

Eine Teilmenge $\Sigma \subseteq FUN \cup PR\ddot{A}$ heißt **Signatur** oder genauer **prädikatenlogische Signatur**. Dies ist gerade die Symbolmenge für die weiteren syntaktischen und semantischen Konstruktionen.

Zu einer gegebenen Signatur Σ bezeichnen wir mit Σ_i die Menge aller i–stelligen Funktions- und Prädikatensymbole aus Σ:

$$\Sigma_i = \{\sigma \in \Sigma \mid |f| = i\}.$$

A.1.2 Terme

Neben den Elementen der Signatur gibt es noch **Variablen**, die Menge aller Variablen werden wir mit VAR bezeichnen, x, y und z werden für beliebige Variablen stehen.

Für eine Signatur Σ ist die Menge $T(\Sigma)$ der Σ–**Terme** induktiv definiert:

Terme:	s, t	$::=$	x	
		$::=$	c	
		$::=$	$f(t_1, \ldots, t_n)$	$f \in \Sigma_n \cap FUN$

Für einen Σ–Term $t \in T(\Sigma)$ ist $\mathcal{V}(t)$ die Menge der in t vorkommenden Variablen.

Die Menge $GT(\Sigma)$ ist die Menge der Σ-**Grundterme**:

$$GT(\Sigma) := \{t \in T(\Sigma) \mid \mathcal{V}(t) = \emptyset\}.$$

A.1.3 Formeln

Zu einer Signatur Σ ist die Menge der Σ–**Formeln** wie folgt induktiv definiert:

Formeln:	φ, ψ	$::=$	$\top \mid \bot$		Wahrheitswerte
		$::=$	$s \doteq t$		Gleichung
		$::=$	$p(t_1, \ldots, t_n)$	$p \in \Sigma_n \cap PR\ddot{A}$	Prädikat
		$::=$	$\neg\psi$		Negation
		$::=$	$\varphi \wedge \psi$		Konjunktion
		$::=$	$\varphi \vee \psi$		Disjunktion
		$::=$	$\varphi \rightarrow \psi$		Implikation
		$::=$	$\exists x\, \varphi$		Existentielle Quantifizierung
		$::=$	$\forall x\, \varphi$		Universelle Quantifizierung

Die Σ–Formeln $\top$ und $\bot$ stellen die "Wahrheitswerte" **wahr** und **falsch** dar. Die Zeichen "$\forall$" und "$\exists$" bezeichnet man als **Quantoren**. Σ–Formeln der Gestalt $\top$, $\bot$, $s \doteq t$ oder $p(t_1, \ldots, t_n$ heißen **atomare Formeln**.

Die Menge der **freien Variablen** $\mathcal{V}(\varphi)$ einer Σ–Formel φ ist induktiv wie folgt definiert:

$$\begin{array}{rcll}
\mathcal{V}(\top) & := & \emptyset & \\
\mathcal{V}(\bot) & := & \emptyset & \\
\mathcal{V}(s \doteq t) & := & \mathcal{V}(s) \cup \mathcal{V}(t) & \\
\mathcal{V}(p(t_1, \ldots, t_n)) & := & \bigcup_{i=1}^{n} \mathcal{V}(t_i) & \\
\mathcal{V}(\neg\varphi) & := & \mathcal{V}(\varphi) & \\
\mathcal{V}(\varphi \odot \psi) & := & \mathcal{V}(\varphi) \cup \mathcal{V}(\psi) & \odot \in \{\wedge, \vee, \rightarrow\} \\
\mathcal{V}(Qx\, \varphi) & := & \mathcal{V}(\varphi) - \{x\} & Q \in \{\forall, \exists\}
\end{array}$$

Falls die Signatur Σ aus dem Kontext heraus ersichtlich ist, oder bei Aussagen, die für beliebige Signaturen gelten, sprechen wir statt von Σ-Formeln oder Σ-Termen auch einfach von Formeln, respektive Termen.

Eine Variable x heißt **gebunden** in der Formel φ, falls in φ $\exists x$ oder $\forall x$ vorkommt. Eine Formel heißt **abgeschlossen** oder **Satz**, falls sie keine freien Variablen enthält, sonst **offen**. Ein Satz, der keine Existenzquantoren enthält nennen wir einen **universellen Satz**.

Sei φ eine mit den freien Variablen $\mathcal{V}(\varphi) = \{x_1, \ldots, x_n\}$. Der **universelle Abschluß** $\forall\,\varphi$ und der **existentielle Abschluß** $\exists\,\varphi$ der Formel φ ist wie folgt definiert:

$$
\begin{aligned}
\forall\varphi &= \forall x_1 \ldots \forall x_n\, \varphi \\
\exists\varphi &= \exists x_1 \ldots \forall x_n\, \varphi
\end{aligned}
$$

Statt $\forall x_1 \ldots \forall x_n\, \varphi$ und $\exists x_1 \ldots \exists x_n\, \varphi$ schreiben wir auch $\forall x_1,\cdots,x_n\, \varphi$ beziehungsweise $\exists x_1,\cdots,x_n\, \varphi$.

A.2 Semantik

Bis jetzt haben wir nur die Syntax der Prädikatenlogik vorgestellt. In diesem Abschnitt wird es darum gehen, die rein syntaktischen Gebilde wie Terme und Formeln mit einer Bedeutung, das heißt einer **Semantik** zu unterlegen.

A.2.1 Strukturen

Eine Struktur $\mathcal{M}$ besteht aus einer nichtleeren Menge M, zusammen mit Funktionen und Relationen auf M. Eine Gruppe $(G, \circ)$ mit der zweistelligen Gruppenoperation "$\circ$" oder eine partiell geordnete Menge $(M, \leq)$ mit der zweistelligen Relation "$\leq$" sind Beispiele für Strukturen. Die zugehörige Signatur gibt an, welche Funktionen und Relationen betrachtet werden.

Definition A.1 (Σ–Struktur) *Sei Σ eine Signatur. Eine Σ–**Struktur** $\mathcal{A}$ ist ein Paar $(A, \mathcal{A}[\![\cdot]\!])$ mit:*

- *A ist eine nichtleere Menge ("Universum" oder "Bereich"). Wir schreiben für A auch manchmal $\mathcal{U}(\mathcal{A})$.*
- *$\mathcal{A}[\![\cdot]\!]$ ordnet jedem n–stelligen Funktionszeichen $f \in \Sigma$ eine n-stellige, totale Funktion*

  $$\mathcal{A}[\![f]\!] : \underbrace{A \times \cdots \times A}_{n\text{-}mal} \to A$$

  zu.
- *$\mathcal{A}[\![\cdot]\!]$ ordnet jedem n–stelligen Prädikatensymbol $p \in \Sigma$ eine n-stellige Relation*

  $$\mathcal{A}[\![p]\!] \subseteq \underbrace{A \times \cdots \times A}_{n-mal}$$

zu.

Die Elemente von $\mathcal{U}(\mathcal{A})$ heißen **Individuen**. Eine Σ–Struktur $\mathcal{A}$, welche nur Funktionen, aber keine Relationen enthält (das heißt $\Sigma \subseteq FUN$), heißt auch Σ–**Algebra**.

Eine Signatur stellt die "Schnittstelle" zwischen Syntax und Semantik dar:

Syntax

$\boxed{\text{Signatur}}$

Semantik

Beispiel A.1 (Natürliche Zahlen) *Die Signatur Σ bestehe aus den zwei Konstanten "$\mathsf{0}$" und "$\mathsf{1}$", und den beiden zweistelligen Funktionssymbolen "$+$" und "$*$". Das Universum der Σ–Algebra $\mathcal{N}$ ist die Menge der natürlichen Zahlen $\mathbb{N}_0$. Die Konstanten "$\mathsf{0}$" und "$\mathsf{1}$" werden als die entsprechenden natürlichen Zahlen, die Funktionszeichen "$+$" und "$*$" als Addition und Multiplikation interpretiert:*

$$Algebra\ \mathcal{N} : \begin{cases} \mathcal{U}(\mathcal{N}) &=& \mathbb{N}_0 \\ \mathcal{N}[\![\mathsf{0}]\!] &=& 0 \\ \mathcal{N}[\![\mathsf{1}]\!] &=& 1 \\ \mathcal{N}[\![+]\!] &=& \textit{Additionsfunktion} \\ \mathcal{N}[\![*]\!] &=& \textit{Multiplikationsfunktion} \end{cases}$$

A.2.2 Auswertung von Termen und Formeln

Eine **Variablenbelegung** nach einer Struktur $\mathcal{A}$ ist eine totale Funktion $\alpha : VAR \rightarrow \mathcal{U}(\mathcal{A})$. Mit $VAL^{\mathcal{A}}$ bezeichnen wir die Menge aller Variablenbelegungen nach $\mathcal{A}$.

Interpretation von Termen

Ein Σ–Term t denotiert zu einer gegebenen Σ–Struktur $\mathcal{A}$ und Variablenbelegung $\alpha \in VAL^{\mathcal{A}}$ ein Element aus dem Bereich $\mathcal{U}(\mathcal{A})$ von $\mathcal{A}$. Dieses Element bezeichnet man mit "$\mathcal{A}[\![\alpha \triangleright t]\!]$" (sprich: "Wert von t in $\mathcal{A}$ unter α").

Zu einer Σ-Struktur $\mathcal{A}$, Variablenbelegung $\alpha \in VAL^{\mathcal{A}}$ und Σ-Term t ist $\mathcal{A}[\![\alpha \triangleright t]\!]$ induktiv definiert:

$$\begin{aligned} \mathcal{A}[\![\alpha \triangleright x]\!] &= \alpha(x) \\ \mathcal{A}[\![\alpha \triangleright f(t_1,\ldots,t_n)]\!] &= \mathcal{A}[\![f]\!]\left(\mathcal{A}[\![\alpha \triangleright t_1]\!],\ldots,\mathcal{A}[\![\alpha \triangleright t_n]\!]\right) \end{aligned}$$

Beispiel A.2 (Natürliche Zahlen) *Sei α eine Variablenbelegung nach $\mathcal{N}$ mit $\alpha(\mathsf{X}) = 2$ und $\alpha(\mathsf{Y}) = 5$. Dann gilt:*

$$\mathcal{A}[\![\alpha \triangleright (\mathsf{X} * (\mathsf{Y} + \mathsf{1}))]\!] = 12$$

Hierbei ist "$(\mathsf{X} + (\mathsf{Y} * \mathsf{1}))$" eine Infixschreibweise für "$+(\mathsf{X}, +(\mathsf{Y}, \mathsf{1}))$".

Der Wert eines Σ-Termes t in einer Struktur hängt nur von den Variablen ab, die in t vorkommen:

Proposition A.1 *Seien Σ eine Signatur, $\mathcal{A}$ eine Σ-Struktur, α, β Variablenbelegungen nach $\mathcal{A}$ und t ein Σ-Term. Falls $\alpha(x) = \beta(x)$ für alle $x \in \mathcal{V}(t)$ gilt, so ist $\mathcal{A}[\![\alpha \rhd t]\!] = \mathcal{A}[\![\beta \rhd t]\!]$.*

Interpretation von Formeln

Zu einer Funktion f bezeichnet $f[y \leftarrow d]$ die Funktion

$$f[y \leftarrow d](x) = \begin{cases} d & \text{falls } x = y \\ f(x) & \text{sonst} \end{cases}$$

Zu einer Σ–Struktur $\mathcal{A}$ spezifiziert eine Σ–Formel φ eine Menge von Variablenbelegungen nach $\mathcal{A}$ (Bezeichnung: $\mathcal{A}[\![\varphi]\!]$), die man als **Lösungen** von φ in $\mathcal{A}$ bezeichnet. Die Lösungen von φ in $\mathcal{A}$ sind gegeben durch:

$$\begin{array}{rcl}
\mathcal{A}[\![\bot]\!] & := & \emptyset \\
\mathcal{A}[\![\top]\!] & := & VAL^{\mathcal{A}} \\
\mathcal{A}[\![s \doteq t]\!] & := & \{\alpha \in VAL^{\mathcal{A}} \mid \mathcal{A}[\![\alpha \rhd s]\!] = \mathcal{A}[\![\alpha \rhd t]\!]\} \\
\mathcal{A}[\![p(t_1, \ldots, t_n)]\!] & := & \{\alpha \in VAL^{\mathcal{A}} \mid (\mathcal{A}[\![\alpha \rhd t_1]\!], \ldots, \mathcal{A}[\![\alpha \rhd t_n]\!]) \in \mathcal{A}[\![p]\!]\} \\
\mathcal{A}[\![\varphi \wedge \psi]\!] & := & \mathcal{A}[\![\varphi]\!] \cap \mathcal{A}[\![\psi]\!] \\
\mathcal{A}[\![\varphi \vee \psi]\!] & := & \mathcal{A}[\![\varphi]\!] \cup \mathcal{A}[\![\psi]\!] \\
\mathcal{A}[\![\neg\varphi]\!] & := & VAL^{\mathcal{A}} - \mathcal{A}[\![\varphi]\!] \\
\mathcal{A}[\![\varphi \rightarrow \psi]\!] & := & \mathcal{A}[\![\neg\varphi \vee \psi]\!] \\
\mathcal{A}[\![\exists x\, \varphi]\!] & := & \{\alpha[x \leftarrow d] \mid \alpha \in \mathcal{A}[\![\varphi]\!], d \in \mathcal{U}(\mathcal{A})\} \\
\mathcal{A}[\![\forall x\, \varphi]\!] & := & \mathcal{A}[\![\neg\exists x\, \neg\varphi]\!]
\end{array}$$

Beispiel A.3 (Natürliche Zahlen) *Eine Lösung der Formel $\mathsf{X} + \mathsf{Y} \doteq \mathsf{X} + 1$ in der Struktur $\mathcal{N}$ ist zum Beispiel eine Variablenbelegung α mit $\alpha(\mathsf{X}) = 1$ und $\alpha(\mathsf{Y}) = 1$.*

Dagegen hat die Gleichung $x + 2 = x + 3$ keine Lösung in $\mathcal{N}$:

$$\mathcal{N}[\![\mathsf{X} + 2 \doteq \mathsf{X} + 3]\!] = \emptyset.$$

Durch diese Semantik genügt es, sich auf einige wenige logische Kombinatoren zu beschränken. So genügen zum Beispiel die Kombinatoren $\neg$, $\wedge$ und $\exists$:

$$\begin{array}{rcl}
\mathcal{A}[\![\bot]\!] & = & \mathcal{A}[\![\neg\exists \mathsf{X}\, \mathsf{X} \doteq \mathsf{X}]\!] \\
\mathcal{A}[\![\top]\!] & = & \mathcal{A}[\![\neg\bot]\!] \\
\mathcal{A}[\![\varphi \vee \psi]\!] & = & \mathcal{A}[\![\neg(\neg\varphi \wedge \neg\psi)]\!] \\
\mathcal{A}[\![\varphi \rightarrow \psi]\!] & = & \mathcal{A}[\![\neg\varphi \vee \psi]\!] \\
\mathcal{A}[\![\forall x\, \varphi]\!] & = & \mathcal{A}[\![\neg\exists x\, \neg\varphi]\!]
\end{array}$$

A.2.3 Gültigkeit und Modelle

Die Relation $\models$

Seien im folgenden Σ eine Signatur, $\mathcal{A}$ eine Σ–Struktur, $\alpha \in VAL^{\mathcal{A}}$, φ eine Σ-Formel.

Die Relation $\mathcal{A}, \alpha \models \varphi$ gilt genau dann, wenn α eine Lösung von φ in $\mathcal{A}$ ist, also:

$$\mathcal{A}, \alpha \models \varphi \qquad :\Longleftrightarrow \qquad \alpha \in \mathcal{A}\,[\![\varphi]\!]\,.$$

Man sagt in diesem Fall auch, daß "φ in $\mathcal{A}$ unter der Belegung α gilt" oder, daß "φ wahr ist in $\mathcal{A}$ unter der Belegung α".

Die Formel φ heißt **erfüllbar in** $\mathcal{A}$, falls φ eine Lösung in $\mathcal{A}$ hat (das heißt $\mathcal{A}\,[\![\varphi]\!] \neq \emptyset$), sonst **unerfüllbar in** $\mathcal{A}$; φ heißt **erfüllbar**, falls es eine Struktur $\mathcal{A}$ gibt, so daß φ erfüllbar in $\mathcal{A}$ ist, sonst **unerfüllbar**.

Falls alle Variablenbelegungen nach $\mathcal{A}$ Lösungen von φ in $\mathcal{A}$ sind, so sagt man auch, daß φ **in** $\mathcal{A}$ **gültig** ist, oder, daß $\mathcal{A}$ **ein Modell von** φ ist (Bezeichnung: $\mathcal{A} \models \varphi$):

$$\mathcal{A} \models \varphi \qquad :\Longleftrightarrow \qquad \mathcal{A}\,[\![\varphi]\!] = VAL^{\mathcal{A}}.$$

Proposition A.2

$$\mathcal{A} \models \varphi \iff \mathcal{A} \models \tilde{\forall}\varphi$$

Falls φ in allen Modellen gültig ist, heißt φ **allgemeingültig**, **gültig** oder eine **Tautologie**. (Bezeichnung: $\models \varphi$):

$$\models \varphi \quad :\Longleftrightarrow \quad \mathcal{A} \models \varphi \qquad \text{für alle } \mathcal{A}.$$

Beispiel A.4 *Gültige Formeln sind zum Beispiel:*

1. $\mathsf{a} \doteq \mathsf{a}$
2. $\mathsf{X} \doteq \mathsf{X}$
3. $(\varphi \wedge (\varphi \rightarrow \psi)) \rightarrow \psi$.

Für Σ–Formeln φ und ψ sagt man, "φ **impliziert** ψ logisch" (Bezeichnung: $\varphi \models \psi$), falls in jeder Σ-Struktur $\mathcal{A}$ die Lösungen von φ auch Lösungen von ψ sind:

$$\varphi \models \psi \quad :\Longleftrightarrow \quad \text{für alle } \Sigma\text{-Strukturen } \mathcal{A}:\ \mathcal{A}\,[\![\varphi]\!] \subseteq \mathcal{A}\,[\![\psi]\!]\,.$$

Sei Φ eine möglicherweise unendliche Formelmenge. Man sagt, daß "Φ die Formel ψ logisch impliziert" oder, daß "ψ eine memlogische Konsequenz von Φ" ist (Bezeichnung: $\Phi \models \psi$), falls für alle Σ-Strukturen $\mathcal{A}$ jede Lösung aller Formeln aus Φ auch Lösung von ψ ist:

$$\Phi \models \psi \quad :\Longleftrightarrow \quad \text{für alle } \Sigma\text{-Strukturen } \mathcal{A}:\ \bigcap_{\varphi \in \Phi} \mathcal{A}\,[\![\varphi]\!] \subseteq \mathcal{A}\,[\![\psi]\!]\,.$$

Eine Σ–Struktur $\mathcal{A}$ heißt **Modell** von Φ (Bezeichnung: $\mathcal{A} \models \Phi$), falls jede Formel aus Φ in $\mathcal{A}$ gültig ist:

$$\mathcal{A} \models \Phi \quad :\Longleftrightarrow \quad \text{für alle } \varphi \in \Phi:\ \mathcal{A} \models \varphi.$$

Beispiel A.5 (Gruppen) *Die Signatur Σ bestehe aus einer Konstanten "e" und einem zweistelligen Funktionssymbol "$\circ$". Die Formelmenge Φ_G besteht aus folgenden Formeln:*

$$\Phi_G : \begin{cases} \forall \mathsf{X} & \mathsf{X} \circ \mathsf{e} \doteq \mathsf{X} \\ \forall \mathsf{X}, \mathsf{Y}, \mathsf{Z} & (\mathsf{X} \circ \mathsf{Y}) \circ \mathsf{Z} \doteq \mathsf{X} \circ (\mathsf{Y} \circ \mathsf{Z}) \\ \forall \mathsf{X} \exists \mathsf{Y} & \mathsf{X} \circ \mathsf{Y} \doteq \mathsf{e} \end{cases}$$

Eine Σ–Struktur $\mathcal{A}$ heißt **Gruppe** *genau dann, wenn $\mathcal{A}$ ein Modell von Φ_G ist. Eine Formel φ ist gültig in allen Gruppen genau dann, wenn $\Phi_G \models \varphi$.*

Folgenden fundamentalen Eigenschaften gelten:

- Die Relation $\varphi \models \psi$ ist im allgemeinen nicht entscheidbar.
- Für eine rekursiv aufzählbare Formelmenge Φ ist die Menge aller logischen Konsequenzen von Φ rekursiv aufzählbar.
- Zu der Struktur der natürlichen Zahlen $\mathcal{N}$ gibt es keine rekursiv aufzählbare Formelmenge $\Phi_\mathcal{N}$, so daß:

$$\text{Für alle Formeln } \varphi : \quad \mathcal{N} \models \varphi \iff \Phi_\mathcal{N} \models \varphi.$$

Zwei Σ–Formeln φ und ψ heißen **logisch äquivalent** (Bezeichnung: $\varphi \models\mid \psi$), falls ihre Lösungsmengen in allen Σ-Strukturen gleich sind:

$$\varphi \models\mid \psi \quad :\iff \quad \text{für alle } \Sigma\text{–Strukturen } \mathcal{A} : \ \mathcal{A}[\![\varphi]\!] = \mathcal{A}[\![\psi]\!].$$

Falls φ von der Form $\cdots \psi \cdots$ ist, wobei ψ selbst wieder eine Formel ist, so heißt ψ **Unter-** oder **Teilformel** von φ.

Proposition A.3 *1. $\models\mid$ ist Äquivalenzrelation auf Formeln.*

2. $\models\mid$ ist Kongruenz auf Formeln, das heißt: Falls φ' aus φ durch Ersetzen einer Unterformel durch eine äquivalente Formel hervorgeht, dann ist: $\varphi' \models\mid \varphi$.

Für eine Σ–Struktur $\mathcal{A}$ heißen Σ-Formeln φ und ψ **$\mathcal{A}$–äquivalent** oder **äquivalent bezüglich $\mathcal{A}$** (Bezeichnung: $\varphi \models\mid_\mathcal{A} \psi$), falls ihre Lösungsmengen in $\mathcal{A}$ gleich sind:

$$\varphi \models\mid_\mathcal{A} \psi \quad :\iff \quad \mathcal{A}[\![\varphi]\!] = \mathcal{A}[\![\psi]\!]$$

Zu einer Formelmenge Φ heißen Σ-Formeln φ und ψ **Φ–äquivalent** oder **äquivalent bezüglich Φ** (Bezeichnung: $\varphi \models\mid_\Phi \psi$), falls sie äquivalent bezüglich aller Modelle von Φ sind:

$$\varphi \models\mid_\Phi \psi \quad :\iff \quad \text{für alle Modelle } \mathcal{A} \text{ von } \Phi\text{: } \varphi \models\mid_\mathcal{A} \psi.$$

Eine Σ-Formel φ **impliziert** eine Σ-Formel ψ **bezüglich einer Σ-Struktur $\mathcal{A}$** (Bezeichnung: $\varphi \models_\mathcal{A} \psi$), falls jede Lösung von φ in $\mathcal{A}$ auch Lösung von ψ in $\mathcal{A}$ ist:

$$\varphi \models_\mathcal{A} \psi \quad :\iff \quad \mathcal{A}[\![\varphi]\!] \subseteq \mathcal{A}[\![\psi]\!].$$

Literaturverzeichnis

[CL88] Hubert Comon and Pierre Lescanne. Equational problems and disunification. Rapport de Recherche 904, INRIA CRIN, Nancy, France, September 1988.

[Cla78] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, NY, 1978.

[Col84] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* Monographs on Pure and Applied Mathematics. North–Holland, 1952.

[Kun87] Kenneth Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[Mah88] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. Research report, IBM, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, U.S.A., 1988.

[Mal71] Anatolií Ivanovič Malc'ev. Axiomatizable classes of locally free algebras of various type. In III Benjamin Franklin Wells, editor, *The Metamathematics of Algebraic Systems: Collected Papers 1936–1967*, chapter 23, pages 262–281. North Holland, 1971.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Science*, 16:158–167, 1978.

[Tar53] Alfred Tarski. *Undecidable Theories.* Studies in logic and the foundations of mathematics. North-Holland, 1953.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

Symbolindex

Anhang A

Index